Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Implementación del protocolo de encaminamiento AODVv2 en Linux.

TITULACIÓN: Grado en Ingeniería Telemática.

AUTOR: Carlos Valdés Pérez

DIRECTOR: Zola, Enrica Valeria / Martín Escalona, Israel

FECHA: 24 de octubre de 2017

Título: Implementación del protocolo de encaminamiento AODVv2 en Linux

Autor: Pérez , Carlos Valdés

Directora: Zola, Enrica Valeria / Martín Escalona, Israel

Data: 24 de octubre de 2017

Resumen

El objetivo del proyecto es la implementación del protocolo AODVv2, como punto de partida para la implementación de un conjunto de protocolos de encaminamiento basados en localización, derivados del protocolo AODVv2.

Este proyecto se enmarca dentro uno de mayor envergadura que comprende las siguientes fases:

1) Implementación de protocolo AODVv2 en un entorno Linux

2) Migración del código para su correcto funcionamiento en dispositivos embedded con microprocesador ARM tales como Raspberry Pi y derivados.

3) Extensión del código realizado para implementar protocolos derivados del AODVv2 original y que empleen la información de localización durante el proceso de encaminamiento.

4) Verificar el rendimiento de dichos protocolos en escenarios reales y contrastarlo con lo previsto por el marco teórico desarrollado con anterioridad

Este documento comprende únicamente la fase 1 arriba indicada y por tanto se centrará en describir los procesos que componen el protocolo AODVv2 y proceder a su implementación.

Title: Implementation of the AODVv2 routing protocol in Linux**.**

Author: Pérez , Carlos Valdés

Director: Zola, Enrica Valeria / Martín Escalona, Israel

Date: October 24th, 2017

Overview

The objective of the project is the implementation of the AODVv2 protocol, as a starting point for the implementation of a set of location-based routing protocols, based on the AODVv2 protocol.

This project is framed within a larger one that includes the following phases:

1) Implementation of AODVv2 protocol in a Linux environment

2) Migration of the code for its correct operation in embedded devices with microprocessor ARM such as Raspberry Pi and derivatives.

3) Extension of the code made to implement protocols derived from the original AODVv2 and that use the location information during the routing process.

4) Verify the performance of these protocols in real scenarios and contrast it with the predicted theoretical framework previously

This document only includes phase 1 above and therefore will focus on describing the processes that make up the AODVv2 protocol and proceed with its implementation.

Frases de agradecimiento:

Silvia, te pido perdón por el tiempo que no he podido estar a tu lado, 10 años son demasiados años.

Manuel Valdés y María Pérez, infinito es el agradecimiento por vuestro apoyo.

Este trabajo va dedicado a todos los profesores que he tenido a lo largo del viaje, a los compañeros que se marcharon y a aquellos que me sufrieron.

Contenido

Lista de acrónimos

- AckReq: Se utiliza en un mensaje de confirmación de respuesta de ruta para indicar que se espera un acuse de recibo.
- AdvRte: Una ruta anunciada en un mensaje de control entrante.
- OrigAddr: La dirección IP de origen del paquete IP que activa el descubrimiento de ruta.
- OrigMetric: El valor de métrica asociado con la ruta a OrigPrefix.
- OrigPrefix: El prefijo configurado en la entrada Router Client que incluye OrigAddr.
- RREP (respuesta de ruta): Tipo de mensaje utilizado para responder a un mensaje de solicitud de ruta.
- RREP_Gen (Generador de RREP): El encaminador que genera el mensaje de respuesta de ruta, es decir, el encaminador configurado con TargAddr como cliente de encaminador.
- RREQ (Solicitud de ruta): Tipo de mensaje utilizado para descubrir una ruta a TargAddr y distribuir la información sobre una ruta a OrigPrefix.
- RREQ_Gen: El Router que genera el mensaje de solicitud de ruta.
- RteMsg (Mensaje de ruta): Un mensaje de Solicitud de ruta (RREQ) o de respuesta de ruta (RREP).
- TargAddr: La dirección de destino de una solicitud de ruta.
- TargMetric: El valor de métrica asociado con la ruta a TargPrefix.
- TargPrefix: El prefijo configurado en la entrada Router Client que incluye TargAddr.
- TargPrefixLen: La longitud del prefijo, en bits, que se configura en la entrada de la tabla Router Client Set que incluye TargAddr.
- TargSeqNum: El número de secuencia del encaminador AODVv2 que originó la respuesta de ruta en nombre de TargAddr.

INTRODUCCIÓN

La finalidad de este proyecto es implementar el protocolo de encaminamiento AODVv2 en código C, y hacerlo correr en una máquina con el sistema operativo LINUX.

AODVv2 es la evolución del protocolo DYMO, y ambos proporcionan el servicio de encaminamiento de paquetes a nivel 3 para redes MANET (Mobile Ad Hoc Network).

Las redes MANET son aquellas que están compuestas por nodos móviles, interconectados por tecnología inalámbrica, donde cualquier nodo puede ser origen o destino de un flujo de datos y ofrece el servicio de encaminamiento de paquetes a nivel 3.

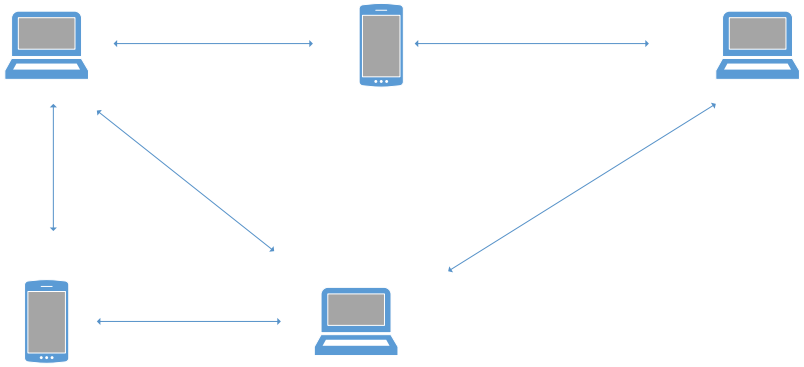


Figura 1 Red MANET formada por portátiles y teléfonos móviles

Una característica más que tienen que cumplir es la ausencia de infraestructura. Esto dota a las redes MANET de una movilidad que permite desplegarlas en situaciones donde la totalidad o parte de la infraestructura de comunicaciones haya quedado inservible, como por ejemplo en el caso de desastres naturales.

Y por último, ya que son redes móviles sin infraestructura, establecer un diagrama de red sería inviable. Por lo que, las redes MANET carecen de ninguna planificación de la red.

Dada la utilidad de este tipo de redes, ya hace varios años que se llevan desarrollando protocolos de encaminamiento específicos para redes MANET, ya que los existentes cuando se desarrolló el concepto de Mobile Ad Hoc Network, principalmente protocolos para redes cableadas (RIP, OSPF), no estaban preparados para entornos dinámicos.

El objetivo principal del proyecto es la implementación del protocolo AODVv2. Este es un objetivo ambicioso y durante el desarrollo se ha tenido que ir reduciendo las expectativas debido a causas que se expondrán en los siguientes capítulos.
El documento está formado por 6 capítulos. En el primero se abordará el estado del arte en protocolos de encaminamiento Ad-Hoc, presentado una clasificación de los más importantes.

En el segundo capítulo se explicará el AODVv2, haciendo hincapié en los procesos que se han implementado y nombrando los que quedan pendientes para trabajos futuros.

El tercer capítulo es el resultado de la investigación sobre las implementaciones existentes que han podido marcar una hoja de ruta para el desarrollo de este proyecto.

En el cuarto capítulo se presentará la solución propuesta que, acompañada de los resultados experimentales del capítulo cinco, fijará el punto de inicio para trabajos futuros.

CAPÍTULO 1. PROTOCOLOS DE ENCAMINAMIENTO AD HOC

Los protocolos de encaminamiento Ad Hoc se pueden clasificar de acuerdo a múltiples parámetros. Sin embargo, históricamente uno de los parámetros más empleados para su clasificación ha sido la manera de proceder a la hora de obtener la ruta entre un origen y un destino. De esta forma, los protocolos de encaminamiento pueden dividirse en tres categorías tal y como se muestra en la figura 2.

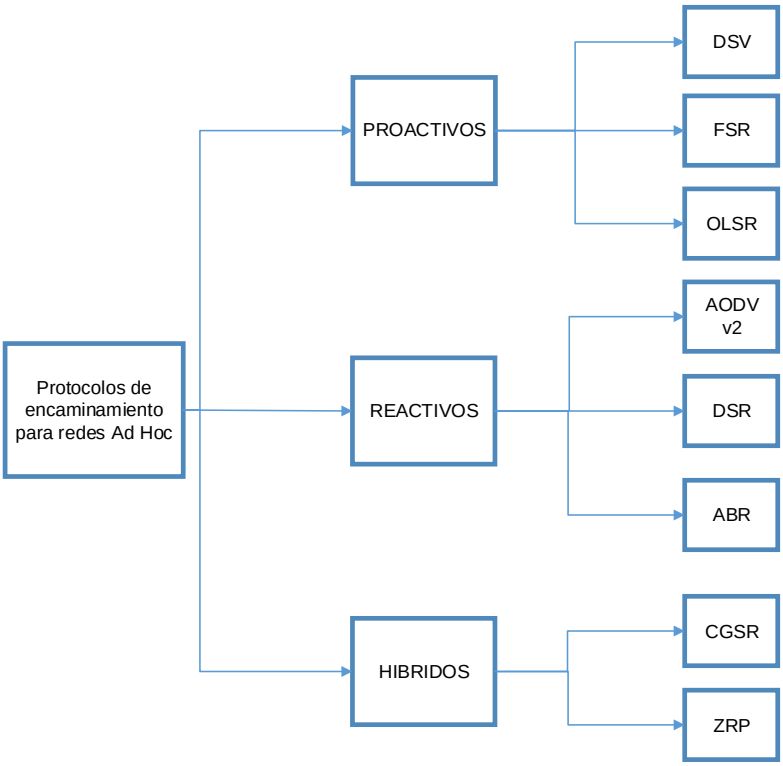


Figura 2 Clasificación protocolos de encaminamiento Ad Hoc

Los de tipo proactivo son aquellos donde las rutas de las tablas de encaminamiento se actualizan periódicamente. Esto permite una latencia baja y constante, pero en contrapartida pueden tener un *overhead* mayor que los reactivos ya que para mantener las tablas precisan del envío de mensajes de control periódicos.

En cambio los protocolos reactivos son aquellos donde las rutas se aprenden y se mantienen solo cuando es necesario. Una ruta para ir de A – B solo se aprende cuando A intenta transmitir a B. Esto hace que el overhead sea menor en comparación con los protocolos proactivos siempre y cuando el número de flujo de datos sea bajo.

El tercer tipo de protocolos, híbridos, busca, como dice su nombre, de unir las ventajas de ambos protocolos precedentes, limitando la aplicación de algoritmos proactivos sólo a los nodos adyacentes del que quiere transmitir.

El objetivo de este trabajo no es detallar el ámbito de los protocolos de encaminamiento Ad Hoc, si bien se ha creído conveniente establecer una mínima clasificación que permita al lector orientarse y entender algunas de las explicaciones que más adelante se proporcionarán.

CAPÍTULO 2. AODVv.2

2.1 Descripción general del protocolo

El protocolo AODVv2 es la evolución del protocolo DYMO. DYMO nace en Julio del 2005 y en 2013 adopta el nombre de AODVv2. En la Figura 3 se puede ver la evolución de las diferentes versiones de DYMO y AODVv2.

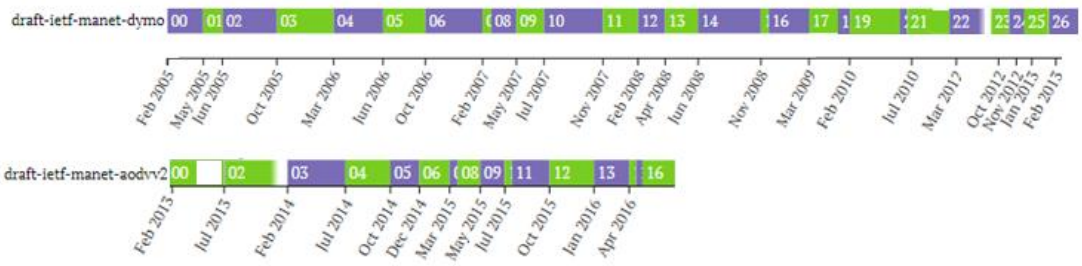


Figura 3. Evolución del protocolo DYMO y AODVv2

Las operaciones básicas son el descubrimiento y mantenimiento de las rutas. Cuando un *router* AODVv2 precisa encontrar un camino hacia un destino, inicia el proceso de descubrimiento de ruta. El nodo origen transmite un *Route Request Message* (RREQ), que contiene la IP del nodo que ha generado el RREQ y la dirección IP o dirección de red para la cual desea encontrar una ruta. Cada host que recibe el RREQ guarda la dirección IP del emisor del paquete RREQ. Por otro lado si el host que recibe el RREQ no es el destino para el cual se inició el descubrimiento de ruta, reenvía el RREQ. Cuando el HOST para el cual se inició el descubrimiento de ruta recibe el mensaje RREQ, envía un *Route Reply Message* (RREP) hacia el host que genero el RREQ, cuando este lo recibe, se da por finalizado el proceso de descubrimiento de ruta.

En la figura 4 se puede ver el proceso de descubrimiento de ruta que inicia el nodo A para encontrar el camino hacia el nodo G. Una vez el nodo G recibe el RREQ, envía el RREP por la ruta que tiene una métrica menor, en este caso la métrica mide el número de saltos.

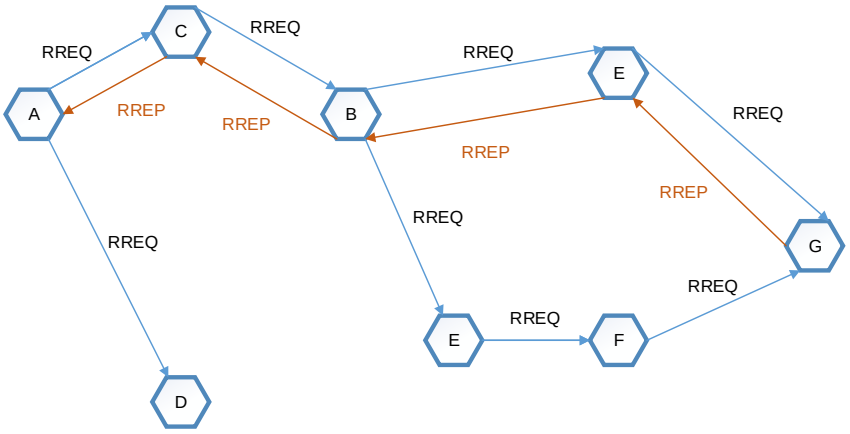


Figura 4. Proceso de descubrimiento de ruta

El mantenimiento de las rutas de las tablas de encaminamiento es el proceso mediante el cual el algoritmo asegura que las rutas activas de la tabla de encaminamiento siguen siendo válidas. Para realizar esta tarea se utiliza los *Route Error Message* (RERR), estos mensajes de control los genera un *router* AODVv2 cuando quiere informar a uno o varios nodos de que una o varias rutas han dejado de ser válidas. Hay tres eventos que provocan él envió de un mensaje RERR:

- Cuando un nodo tiene que reenviar un paquete IP pero no existe una ruta válida en su tabla de encaminamiento. En este caso el nodo enviara un RERR a la fuente para informar que no existe una ruta haca el destino.

- Cuando no se puede reenviar un mensaje RREP porque la ruta hacia el generador del RREQ no es válida. En este caso el nodo debe enviar en RERR hacia el generador del mensaje RREP para infórmale que la ruta hacia el origen del mensaje RREQ no es válida.

- Cuando un nodo detecta que uno de los enlaces que un vecino se ha roto, debe informar a todos los nodos que usan ese enlace de todas las rutas que han pasado a ser invalidas.

Hasta ahora se ha presentado una descripción general del funcionamiento del protocolo AODVv2. En los sucesivos apartados de este capítulo se entrará en profundidad en los procesos, estructura de datos y mensajes implementados y testados en escenarios reales. Estos procesos se enmarcan en el procedimiento de descubrimiento de la ruta. Dicha operación engloba varios procesos y hace uso de estructuras de datos para almacenar entre otras la información de las rutas. Si bien existen algunos procedimientos no implementados (habida cuenta de la limitación temporal del proyecto), el descubrimiento de la ruta comprende en esencia la mayor parte del protocolo AODVv2, por lo que constituye un núcleo consistente sobre el que avanzar en trabajos futuros.

2.2 Estructuras de datos

Las estructuras de datos son las clases en las que se organiza la información y a la cual tendrán acceso los distintos procesos que componen el protocolo AODVv2. Para consultar el detalle de los campos que componen cada estructura de datos dirigirse al anexo A

2.2.1 Interfaceset

La *interfaceset* contiene información relativa a las interfaces inalámbricas del encaminador que usará el protocolo AODVv2. Tiene dos características importantes:

- Se pueden utilizar múltiples interfaces inalámbricas en un solo encaminador.

- Se pueden configurar múltiples interfaces inalámbricas de un encaminador con la misma dirección IP.

2.2.2 Router Client Set

Un *router* AODVv2 solo ofrece el servicio de descubrimiento de rutas a sus aplicaciones locales, y a los clientes registrados en el “Router Client Set”, y solo generará mensajes RREQ y RREP en nombre de los clientes que en ella estén registrados.

2.2.3 Neighbor Set

La tabla *Neighbor Set* contiene información relativa a los routers vecinos. Esta se actualiza a partir de los mensajes de control. También contiene información relativa a la bidireccionalidad del enlace, una ruta solo se considerará valida cuando se confirme que el enlace es bidireccional.

2.2.4 Sequence Number

Los números de secuencia permiten a los encaminadores AODVv2 determinar el orden temporal de los mensajes de descubrimiento de ruta, identificando la información de enrutamiento obsoleta para que pueda descartarse.

Cada router AODVv2 debe mantener su propio *Sequence Number*, este se incluye en todos los mensajes RREQ y RREP creados por él.

2.2.5 Local Route Set

Todos los routers AODVv2 deben mantener la Local Route Set, esta contiene información sobre las rutas aprendidas a partir de los mensajes de control.

Cuando una ruta se considere válida se deberá añadir la entrada correspondiente en la tabla de encaminamiento, y cuando una ruta pasa de válida a inválida se debe borrar la entrada correspondiente en la tabla de encaminamiento.

2.2.6 Multicast Route Message Set

Los mensajes RREQ (Route Request) por defecto son multicast y estos pueden ser reenviados varias veces. El *multicast route message set* tiene como finalidad proporcionar información relativa a los mensajes RREQ y RREP que han sido recibidos previamente, y de esta manera poder compararlos con los mensajes de ruta recibidos y determinar si la información que contienen es antigua. Esto permite al router controlar el envío de tráfico redundante.

2.3 Mensajes

En este apartado se definen los mensajes de control que el protocolo utiliza para comunicar entre nodos información relativa a las rutas. AODVv2 define 4 tipos de mensajes de control:

- Route Request (RREQ)
- Route Reply (RREP)
- Route Reply Acknowledgement (RREP_Ack)
- Route Error (RERR)

En este proyecto solo se han implementado los que intervienen en el proceso de descubrimiento de ruta, RREQ, RREP y RREP_Ack. Para obtener información relativa a los mensajes RERR consultar el documento *draft-ietf-manet-aodvv2-16* [1].

2.3.1 Route Request (RREQ)

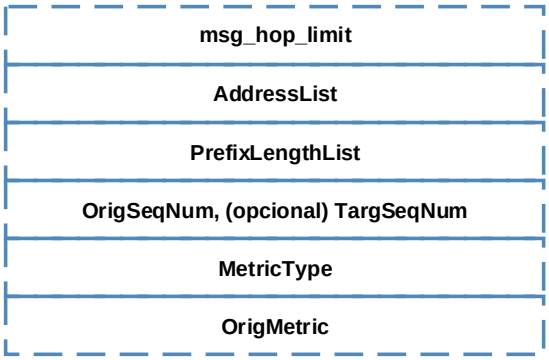


Figura 5 Contenido del mensaje RREQ

- msg_hop_limit: Contiene un número entero que decrece en 1 cada salto que atraviesa el mensaje RREQ. El RREQ_Gen establece el número máximo de saltos que atravesará el mensaje RREQ.
- AddressList: Contiene OrigPrefix y TargPrefix
- PrefixLengthList (Opcional): Contiene OrigPrefixLen, si se omite, la longitud del prefijo (en bits) es igual a la longitud de la dirección OrigAddr.
- OrigSeqNum : Número de secuencia de OrigPrefix
- MetricType: Tipo de métrica asociada con OrigMetric.
- OrigMetric: El valor de la métrica asociada a la ruta a OrigPrefix

2.3.2 Route Reply (RREP)

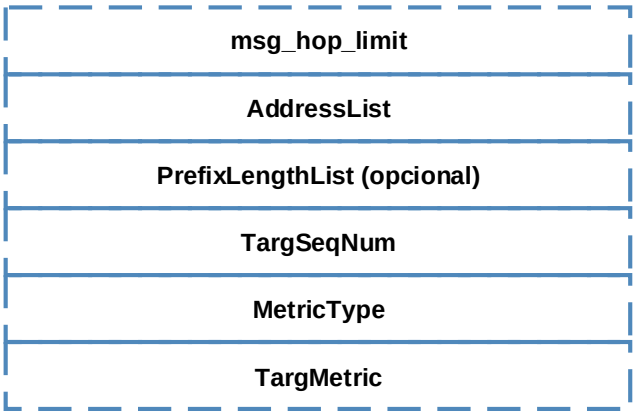
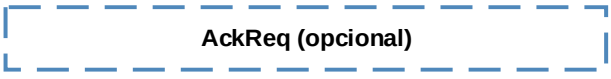


Figura 6 Contenido del mensaje RREP

- msg_hop_limit: Contiene un número entero que decrece en 1 cada salto que atraviesa el mensaje RREP. El RREP_Gen establece el número máximo de saltos que atravesará el mensaje RREP.
- AddressList: Contiene las direcciones OrigPrefix y TargPrefix.
- PrefixLengthList (opcional): Contiene TargPrefixLen, si se omite, la longitud del prefijo (en bits) es igual a la longitud de la dirección TargAddr.
- TargSeqNum: El número de secuencia asociado a TargPrefix.
- MetricType: El tipo de métrica asociada a TargMetric.
- TargMetric: El valor de la métrica asociada con la ruta a TargPrefix.

2.3.3 Route Reply Acknowledgement (RREP_Ack)



- AckReq (Opcional): Si se incluye, informa al receptor debe enviar un RREP_Ack response para confirmar la bidireccionalidad del enlace.

2.4 Procesos

En este apartado se va a dar una breve descripción de los procesos que se han podido implementar y probar. Para obtener una descripción más detallada consultar el anexo A.

2.4.1 Next Hop Monitoring

Este proceso tiene como finalidad asegurar que no se establecen rutas a través de enlaces unidireccionales, para ello los routers AODV2 deben verificar la bidireccionalidad del enlace con el siguiente salto antes de marcar una ruta como válida en el *local route set*.

- Para comprobar si un enlace es bidireccional con un router *upstream* se utiliza el mensaje de control *Route Reply Acknowledgement* (RREP_Ack). Al enviar un RREP_Ack, se espera un RREP_Ack como respuesta, si este llega en un tiempo menor a RREP_Ack_SENT_TIMEOUT demuestra que el enlace es bidireccional, en caso contrario el enlace se considera unidireccional.

- Para un router *downstream*, el hecho de recibir un mensaje RREP que contiene en el campo TargAddr la dirección destino de una solicitud de ruta, es una confirmación de que el enlace está activo y es bidireccional, ya que, un mensaje RREP requiere que un mensaje RREQ previamente haya recorrido el enlace en dirección contraria.

2.4.2 Neighbor Set Update

Este proceso tiene como finalidad de actualizar la tabla Neighbor Set. Cuando se recibe un mensaje de control se inicia el proceso para actualizar la tabla *Neighbor Set*, esto permite registrar los vecinos del router AODVv2 y establecer la relación que mantiene con cada una de ellos.

Cuando un router recibe un mensaje RREP y se esperaba su recepción, el enlace con el router que ha enviado el paquete es confirmado como bidireccional, y por lo tanto el estado de la entrada correspondiente de la Neighbor Set cambia a *Confirmed.*

Cuando un router recibe un mensaje RREP_Ack y este es debido al envío de un RREP_Ack con AckReq. El enlace es confirmado como bidireccional y se tiene que actualizar la tabla Neighbor Set.

2.4.3 Procesado de la información de los mensajes de ruta

En todos los mensajes de ruta hay información sobre una ruta, los RREQ contienen la ruta hacia OrigPrefix, y los RREP hacia TargPrefix. Esta información se almacena en *Local Route Set*.

Como paso previo al proceso de evaluación, se convierten las estructuras de los mensajes RREQ y RREP a una estructura tipo AdvRte, común para ambos, esto facilita el proceso de desarrollo reduciendo el número de funciones a implementar.

2.4.4 Evaluación de la información de ruta

Este proceso tiene como finalidad evaluar si la información de la ruta que contiene el AdvRte se utilizará para actualizar la tabla *Local Route Set,* para ello se compara el coste y el número de secuencia del AdvRte con la entrada correspondiente en la tabla *Local Route Set*.

2.4.5 Actualización de la información de las rutas

Después de determinar que el AdvRte se utilizará para actualizar *Local Route Set*, este proceso se encarga de añadir una nueva entrada en la Local Route Set o actualizar una existente.

2.4.6 Eliminación de los mensajes redundantes usando la Multicast Route Message Set

Cuando los mensajes de ruta inundan una MANET, un nodo podría recibir varias veces el mismo mensaje de ruta, si no se evita, parte de estos mensajes serán reenviados generando trafico innecesario.

Para solucionar este problema cada router AODVv2, almacena información de los mensajes de ruta que recibe en la tabla *Multicast Route Message Set.*

Cada vez que se recibe un mensaje RREQ o RREP, se consulta en la tabla *Multicast Route Message Set* si la información que contiene el mensaje entrante es redundante o no.

A partir de esto se toma la decisión si el mensaje es reenviado o no.

2.4.7 Creación de mensajes RREQ

Un mensaje RREQ se genera cuando un *Client Router* registrado en la tabla *Local Route Set* de un router AODVv2 quiere enviar un paquete IP y no existe una ruta hacia al destino en su tabla RIB. Tras configurar los parámetros descritos en el apartado 2.3.1 se procede a su envío. La dirección IP de destino del paquete que contiene el mensaje RREQ será la dirección multicast 224.0.0.109.

2.4.8 Recepción de mensajes RREQ

Este proceso se encarga de realizar las operaciones a efectuar cuando un router AODVv2 recibe un mensaje RREQ. Entre ellas chequea el contenido de los campos para comprobar que son válidos, actualiza las tablas Neighbor Set, *Local Route Set* y *Multicast Route Message Set*. Y por último si la solicitud de

descubrimiento de ruta va dirigida a él, envía un mensaje RREP. Si no es así reenvía el mensaje RREQ.

2.4.9 Reenvío de mensajes RREQ

Un mensaje RREP se genera cuando un nodo recibe un RREQ y el campo AdressList.TargPrefix del mensaje coincide con una entrada de la tabla Router Client Set del router. Cuando esto sucede genera un mensaje RREP configurando los campos descritos en el apartado 2.3.2 y lo envía en dirección al RREQ_Gen.

2.4.11 Recepción de mensajes RREP

Este proceso se encarga de realizar las operaciones a efectuar cuando un router AODVv2 recibe un mensaje RREP. Entre ellas chequea el contenido de los campos para comprobar que son válidos, actualiza las tablas Neighbor Set, *Local Route Set* y *Multicast Route Message Set*.

Sí el destino final del mensaje es el propio router, y el mensaje contiene una ruta valida se da por finalizada el proceso de descubrimiento de ruta, añadiendo la entrada correspondiente a la tabla de encaminamiento.

2.4.12 Reenvío de mensajes RREP

Este proceso tiene como finalidad el reenvío de los mensajes RREP, para ello comprobará si no se ha superado el número de saltos máximo, si es así reenvía el mensaje.

2.4.13 Generación de mensajes RREP_Ack Request

Un mensaje RREP_Ack será generado si un mensaje RREP se envía por un enlace del cual se desconoce si es bidireccional.

El RREP_Ack Request se enviará a LocalRoute [OrigPrefix].NextHop a través de LocalRoute [OrigPrefix].NextHopInterface.

La entrada para LocalRoute [OrigPrefix].NextHop en la tabla Neighbor Set se actualizará siguiendo el proceso definido en el apartado 2.4.2.

2.4.14 Recepción de mensajes RREP_Ack

Cuando un router AODVv2 recibe un RREP_Ack, comprobará si el mensaje contiene un AckReq y si el mensaje era esperado o no. Sí el mensaje contiene

un AckReq iniciara el proceso para enviar un RREP_Ack Response, si no es así y el mensaje era esperado actualizará la tabla Neighbor Set para establecer el enlace con el emisor del RREP_Ack como bidireccional.

2.4.15 Generación de RREP_Ack Response

Un router AODvv2 generará un RREP_Ack *Response* cuando reciba un RREP_Ack que contenga un AckReq.

2.5 Formato de paquetes para redes MANET

AODVv2 especifica en su draft que los mensajes de control tienen que mapearse en un contenedor llamado *Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format* [RFC5444 [3]]. Este formato de paquete proporciona un encapsulado único para múltiples protocolos de encaminamiento Ad Hoc.

El RFC5444 [3] dota de una mayor eficiencia a las trasmisiones de los mensajes de control, estructura de tal manera el contenido que reduce el número de bytes a transmitir.

El formato RFC5444 [3] define los siguientes elementos:

- Paquete: Es la entidad de mayor nivel. Un paquete contiene una cabecera y cero o más mensajes.
- Mensaje: Es la entidad que transporta la información del protocolo. Un mensaje está formado por una cabecera, un bloque TLV y un bloque de direcciones.
- Bloque de direcciones: Está formado por una o más direcciones, y un bloque de atributos.
- Bloque TLV: Está formado por uno o más TLV.
- TLV: Es una estructura que tiene la forma “type-length-value”. Donde
    - Type: Es el identificador del tipo de dato que viene a continuación.
    - Length: Este campo indica cuantos bytes ocupa el campo value.
    - Value: Es el valor concreto del objeto al que se refiere.

En la figura 7 se puede ver representado la estructura completa del paquete RFC5444 [3] y sus dependencias:

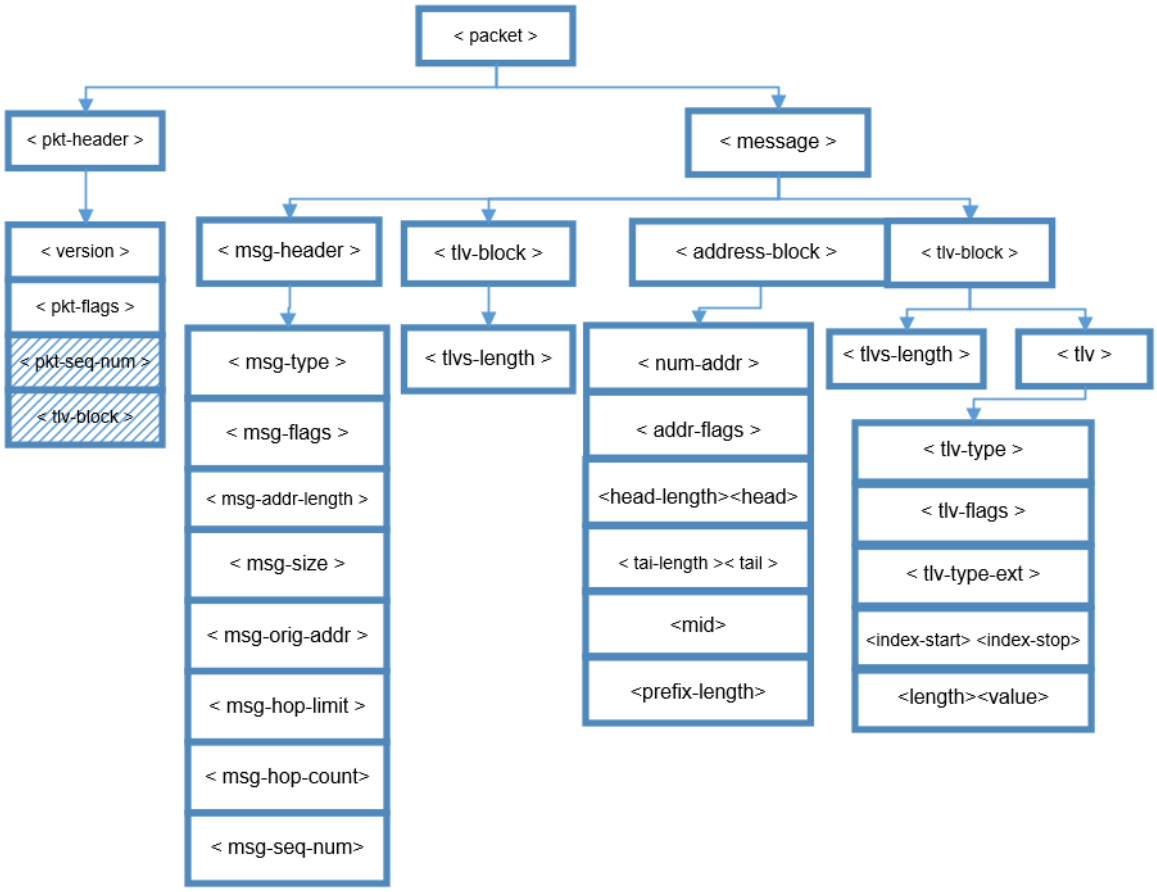


Figura 7. Estructura del paquete RFC5444 [3]

Cada tipo de mensaje de control se tiene que adaptar al formato del paquete RFC5444 [3]. A modo de ejemplo, se puede encontrar en el anexo C un ejercicio teórico en el cual se adapta un mensaje RREQ al RFC5444 [3].

CAPÍTULO 3. TRABAJOS ANTERIORES

En este capítulo se presentan los resultados obtenidos tras la búsqueda de implementaciones ya realizadas de protocolos de encaminamiento Ad Hoc, empleando el lenguaje C y que se ejecuten en un sistema operativo LINUX.

Que cumplan los requisitos anteriores solo se ha podido encontrar una. Se trata de la tesis realizada por el señor Rolf Ehrenreich ThorupImp, "*Implementing and evaluating the DYMO routing protocol*" [4]. Una vez leído el documento de dicha tesis y analizado el código detenidamente, no ha sido posible usar su trabajo como punto de partida de este proyecto, debido a múltiples razones:

- La solución propuesta implica la modificación del kernel de Linux. La versión de kernel sobre la que Rolf Ehrenreich trabaja es la 2.6. Portar el código desde dicha versión a la versión actual del kernel (4.13) es un trabajo arduo, sin garantía de éxito (por las diferencias notables entre ambas versiones de kernel) y que requiere de unos conocimientos profundos de la arquitectura del sistema, fuera de lo que es la formación actual recibida. Por lo tanto se considera tomar este trabajo como punto de partida sería tan trabajoso como iniciar el desarrollo desde cero.

- La solución está distribuida en varios módulos del kernel, lo que dificulta su comprensión y verificación. Teniendo en cuenta las limitaciones temporales del proyecto, no se podría abordar a tiempo su estudio y posterior modificación.

- Parte de la implementación está realizada en código LUA, lenguaje del que no se dispone de formación en este instante.

Tal como se describe en el documento AODV Routing Protocol Implementation Design [2], hay tres estrategias de diseño para abordar la implementación del protocolo AODVv2:

- *Snooping*

- Modificación del kernel de Linux

- Netfilter

En los próximos apartados se detallarán cada una de ellas, mostrando sus ventajas y desventajas.

3.1 *Snooping*

La traducción de *Snooping* es fisgonear, ¿el qué?, los paquetes entrantes y salientes. Esta técnica permite saber cuándo el HOST local ha generado un paquete ARP para descubrir la dirección de la capa MAC, eso significa que no conoce una ruta hacia el destino por lo que se iniciaría el descubrimiento de la ruta.

La ventaja más importante es que no requiere programar el kernel ya que el código para realizar Snooping ya está integrado en el. Esto permite una instalación y ejecución simple.

El problema de usar esta técnica es un abuso y dependencia del protocolo ARP, que puede provocar que las tablas de enrutamiento y la memoria cache de ARP no se sincronicen.

3.2 Modificación del Kernel de Linux

Esta técnica se basa en usar una ventaja del sistema operativo de Linux, que es cargar módulos de sistema en tiempo de ejecución. Estos módulos permiten conectar eventos de sistema con un programa en el espacio de usuario.
Los módulos permiten conocer, por ejemplo, cuando el sistema desconoce una ruta hacia un destino, esto generaría una interrupción en el código del espacio de usuario, inmediatamente se iniciaría el descubrimiento de la ruta.

La principal ventaja es que la velocidad de los procesos que se ejecutan en el espacio del Kernel, pero el código para generar los módulos, aun siendo C, es muy complejo y no está a mi alcance, de momento.

3.3 Netfilter

Netfilter es un programa de filtrado de paquetes implementado en el Kernel de Linux desde la version 1.1. Este programa permite leer las cabeceras de los paquetes IP y ejecutar políticas de actuación. Iptables es el programa del espacio de usuario que permite interactuar con Netfilter, generando nuevas reglas, políticas, en definitiva, tomando decisiones que permiten descartar paquetes, modificar sus cabeceras y reenviarlos por una interfaz de red.

La gran ventaja de Netfilter son los *hook-point*, básicamente son 5 puntos dentro del Kernel de Linux donde se pueden añadir reglas de procesado de paquetes.

La figura 8 muestra la localización de estos puntos:

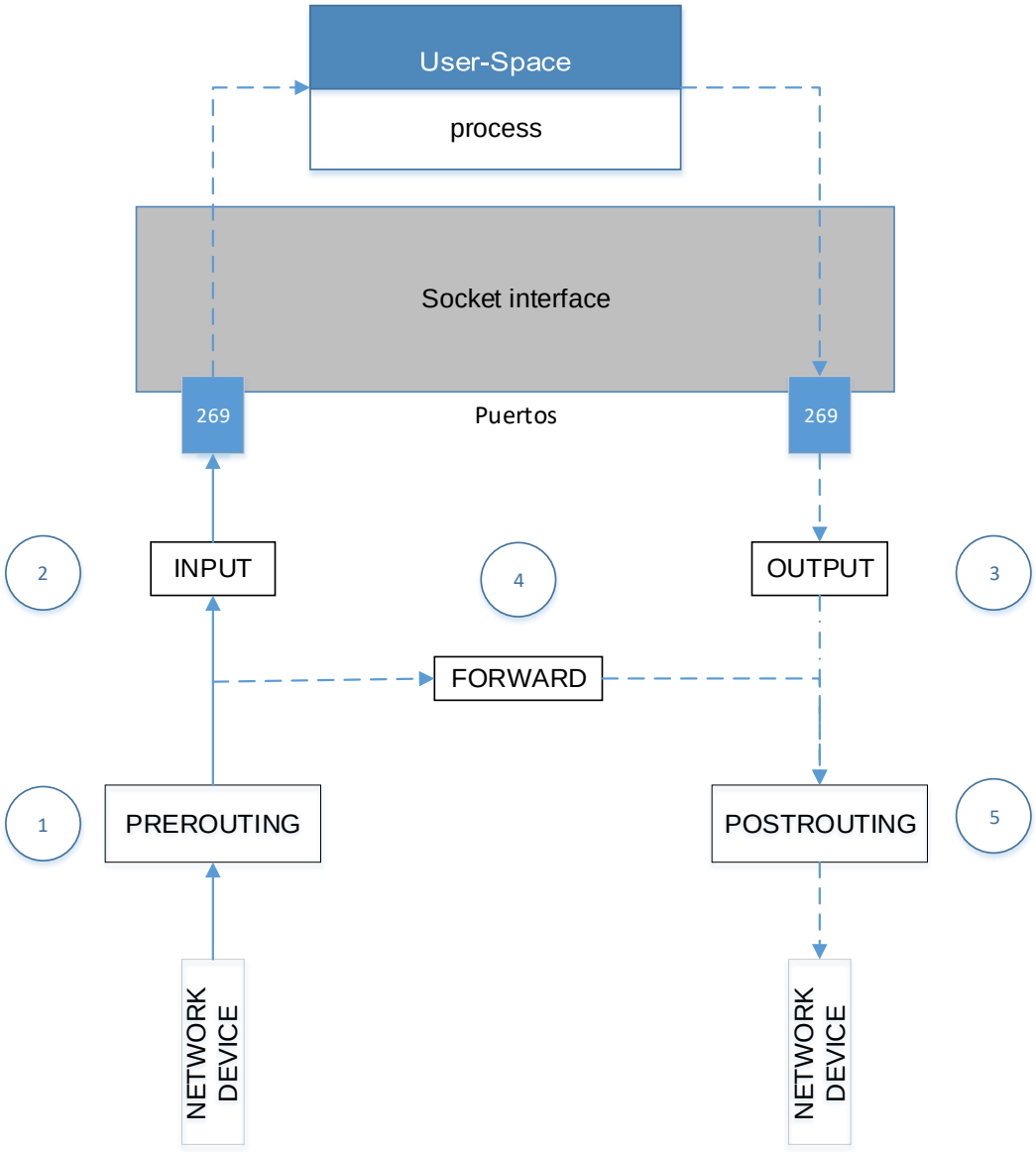


Figura 8. Esquema de los hook-point de Netfilter

- En el hook-point de PREROUTING (1) pasan todos los paquetes recibidos por la interfaz.

- Por INPUT (2) pasan todos los paquetes destinados al HOST.

- Por FORWARD (4) pasan aquellos paquetes que no van dirigidos al HOST y son encaminados hacia otra máquina, es decir cuando cumple funciones de router.

- Por OUTPUT (3) pasan todos los paquetes generados por algún proceso local.

- Por POSTROUTING (5) pasaran todos los paquetes encaminados hacia el exterior, tanto si los ha generado la propia maquina como si no.

Son en estos puntos donde se pueden añadir reglas de procesado, establecer políticas y acciones. Es el conjunto de acciones (target) lo que permite, básicamente, aceptar (ACCEPT), descartar (DROP) y encolar un paquete (QUEUE).

El target QUEUE es el pilar central de esta estrategia de diseño, permite enviar un paquete a una cola definida en el espacio de usuario de Linux, y ahí es donde estará corriendo el proceso AODVv2, el cual tomará las decisiones de encaminamiento.

Para facilitar la comunicación entre Netfilter y el proceso del espacio de usuario tenemos la librería *libnetfilter_queue,* un API con las funciones necesarias que permite obtener los paquetes de la cola y ejecutar un veredicto sobre él.

En la página web de los desarrolladores de Netfilter proporcionan un ejemplo de código que realiza las siguientes funciones:

- Empareja una cola definida por el usuario con una función de *Callback* .
- Imprime por consola las cabeceras de los paquetes de la cola.
- Ejecuta el veredicto ACCEPT sobre todos los paquetes de la cola.

Y ese código es el punto de partida en el cual se basa la solución para la implementación del protocolo AODVv2 de este proyecto.

CAPÍTULO 4. SOLUCIÓN PROPUESTA

En este capítulo se presenta la solución obtenida a la implementación del protocolo AODVv2. La implementación se ha desarrollado en lenguaje c por los siguientes motivos:

1. Acceso y uso de punteros. El uso de punteros optimiza al máximo la memoria al acceder a ella a un nivel muy bajo.

2. Lenguaje eficiente. Es el lenguaje que mejor aprovecha la CPU de la máquina, sin tener las desventajas de los lenguajes ensambladores.

3. Operadores a nivel de bit. Para este proyecto era necesario efectuar operaciones bit a bit.

El sistema se ha diseñado con estas directrices:

- Todos los nodos que formaran la MANET tienen calidad de router y ejecutan el protocolo AODVv2.

- El flujo de datos solo lo generarán procesos internos de cada máquina, por ejemplo el programa PING.

- Un nodo no podrá iniciar el proceso de descubrimiento de una ruta si tiene uno ejecutándose, hasta que no finalice la primera instancia no podrá iniciarse una segunda.

- El plano de reenvío está separado del plano de control. Una vez se agreguen las rutas en la tabla RIB, será el Kernel quien se encargue de encaminar los paquetes IP. De esta manera el sistema no vera mermado su rendimiento a causa del encaminamiento.

- Inicialmente la tabla de encaminamiento estará vacía. Es AODVv2 la única entidad que tendrá acceso a las operaciones de consulta, agregar o eliminar una ruta. De esta manera la tabla *Local Route Set* estará sincronizada con la tabla RIB.

- Todas las direcciones IP de los nodos de la MANET pertenecerán a la misma red.

- Por falta de tiempo los mensajes RREQ se enviaran a la dirección de Broadcast de la IP que esté configurada en la interfaz de salida, en vez de la dirección multicast como indica el protocolo

El código se puede dividir en dos bloques:

- Plano de datos: engloban los archivos y funciones que conectan Netfilter con el algoritmo AODVv2 del espacio de usuario. Está formado principalmente

por el Kernel de Linux, este es la base donde se sustenta el algoritmo AODVv2. Para que AODVv2 pueda procesar los mensajes de control y ejecutar el descubrimiento de una ruta es necesario “un circuito de adaptación”. Este está formado:

- El script “*Confignetfilter.sh*”. Este contiene las reglas para que Netfilter pueda encolar los mensajes de control recibidos por la interfaz de red, y el flujo de datos generado por un proceso interno de la máquina.

- Dos hilos de ejecución concurrente para que el sistema pueda procesar en el mismo instante de tiempo los mensajes de control y el algoritmo de descubrimiento de una ruta.

- Una función de *Callback* que permita ejecutar una función para cada uno de los paquetes de la cola, y emitir un veredicto sobre ellos.

- Plano de control: Engloban todas las funciones y archivos que tienen como función cumplir los objetivos de algoritmo AODVv2 definido en el draft. Este está formado por:

  - Data_Set: Engloba las estructuras datos definidas en el apartado 2.2. El nombre de las variables que forman las estructuras el mismo que usa para definirlas en el draft.

  - aodvv2_message: Engloba los mensajes de control implementado, RREQ, RREP y RREP_Ack, y el código sin comprobar del RERR. El nombre de las estructuras que forman la entidad mensaje coincide con el enumerado en el draft.

  - Linux: Engloba las funciones que realizan consultas al sistema operativo y que se deberían revisar si se tiene la intención de probar el código en una version de kernel superior a la 4.4

  - Process: Engloba todos los procesos definidos en el apartado 2.4. El nombre del archivo y de la función principal de cada uno coincide con el enumerado en el DRAFT.

  - RFC5444 [3]: Engloba los archivos que contienen las funciones que adaptan el formato de mensajes de AODVv2 al RFC5444 [3].

4.1 Plano de datos

4.1.1 Entrada al sistema

Para que el protocolo de encaminamiento AODVv2 pueda decidir sobre la vida de un paquete es necesario que este sea enviado al espacio de usuario, de esto se encarga dos simples reglas que se instalan en Netfilter cuando el protocolo se inicia. Son las siguientes:

- *-A INPUT -p UDP --sport 269 --dport 269 -j NFQUEUE --queue-num 0*
- *-A OUTPUT -j NFQUEUE --queue-num 1*

La primera regla, tiene como acción enviar a la cola número 0 los paquetes UDP con puerto origen y destino 269. Este es el puerto UDP que deben usar los mensajes de control en las redes MANET [RFC5498]

La segunda regla envía los paquetes que pasen por OUTPUT a la cola número 1. Con esta maniobra conseguimos gestionar los paquetes generados por procesos locales.

Estas reglas junto con otras que se nombraran más adelante se encuentran el script *confnetfilter.sh, q*ue se ejecuta en la función “main (int argc, char *argv [])” del archivo “main.c”.

Para que el kernel de Linux ofrezca el servicio de encaminamiento hay que activarlo. Para ello incluimos en archivo *confnetfilter.sh* la siguiente línea.

- *echo 1 > /proc/sys/net/ipv4/ip_forward*

4.1.2 Hilos de ejecución

Para que el sistema pueda procesar en el mismo instante de tiempo los mensajes de control y el algoritmo de descubrimiento de una ruta es necesario hacer uso de la concurrencia.

En la función *main ()* se definen dos hilos de ejecución:

- *pthread_create(&thread_internal_id,&attr,thread_internal,(void*)&data_pointers)*
- *pthread_create(&thread_external_id,&attr,thread_external,(void*)&data_pointers)*

El hilo definido por el identificador *thread_internal_id,* llama a la función *thread_internal.* Este es el hilo donde se ejecutan los procesos encargados de gestionar los paquetes de la cola número 1.

El hilo definido por el identificador *thread_external_id,* llama a la función *thread_external.* Este es el hilo donde se ejecutan los procesos encargados de gestionar los paquetes de la cola número 0.

4.1.3 Funciones de *Callback*

La traducción de *Callback* es, devolución de llamada, en el ámbito de la programación se utiliza para definir a una función A que se ejecuta cuando se llama a una función B. La API de Netfilter proporciona una función Callback, esta se ejecutará para cada paquete de la cola.

En la función *void* thread_internal (void *arg)* del archivo *InternalQueue*.c se define la función que asocia la cola número 1 con la función de *Callback* identificada con el nombre "*callback_internal*".

- *nfq_create_queue(h, 1, &callback_internal, data_pointers);*

En la función *void* thread_internal (void *arg)* del archivo *ExternalQueue*.c se define la función que asocia la cola número 0 con la función de *Callback* identificada con el nombre "*callback_external*".

- *nfq_create_queue(h, 0, &callback_external, data_pointers);*

Para procesar los paquetes de la cola necesitamos el socket asociado y entrar en un bucle de recepción. Esta tarea la cubren las funciones *nfq_fd ()*, para obtener la descripción del archivo y *recv ()* para entrar en modo recepción. Se ejecutan dentro de la función *void *thread_external (void *arg)*, y *void *thread_internal (void *arg).* El código tiene la siguiente forma:

- fd = nfq_fd (h);

- while ((rv = recv (fd, buf, sizeof(buf), 0)) && rv >= 0){
  }

4.1.4 Acciones sobre paquetes

Tal como se ha mencionado en el apartado 3.3, desde el espacio de usuario se efectuara un veredicto sobre los paquetes que están en la cola. En este proyecto se implementa el concepto de veredicto de dos maneras:

1. En la primera se hace un uso típico de filtrado de paquetes tal como se muestra en la figura 9:

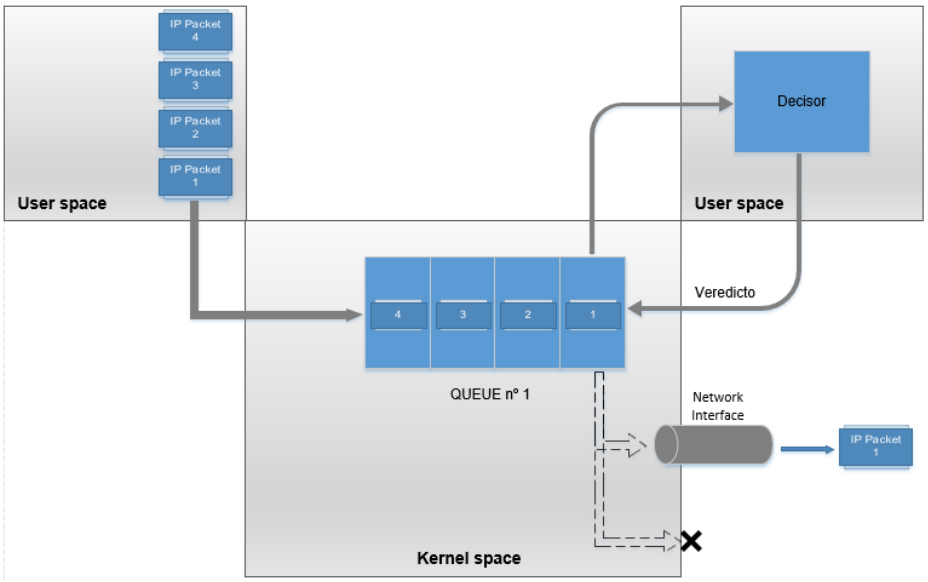


Figura 9. Esquema de sistema de filtrado de paquetes en el espacio de usuario

Un proceso interno de la maquina genera un flujo de datos, estos se encolan en la QUEUE nº1, se envía una copia del primer paquete de la cola al decisor, este en base a unos criterios emite un veredicto:

- ACCEPT: El paquete abandona la cola y se envía por la interface de red hacia su destino.

- DROP: El paquete abandona la cola y se elimina.

  Esta manera de proceder esta implementado en la función *callback_internal ()* del archivo *InternalQueue.c.*

- int callback_external (struct nfq_q_handle *qh, struct nfgenmsg *nfmsg, struct nfq_data *nfa, void *data){

```
if (veredict_flag == 0)
 {
 return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
 }
else
 {
 return nfq_set_verdict(qh, id, NF_DROP, 0, NULL);
 }
}
```

  El decisor lo implementa la función external_process () del archivo *InternalQueue.c.* Esta función se ejecuta desde callback_internal () y devuelve el identificador de paquete que se ha procesado y el veredicto dictaminado

- int callback_external(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,struct nfq_data *nfa, void *data){

```
 int id=external_process (nfa, data_pointers, &veredict_flag);
}
```

2. La segunda manera de proceder afecta solo a los paquetes que se encolan en la QUEUE nº 0. Estos paquetes solo pueden ser del tipo UDP y transportar los mensajes de control AODVv2, esto se debe a las directrices enumeradas al inicio del capítulo 4.

   Los mensajes de control transportan información sobre una ruta, cuando llegan al nodo al cual van dirigidos han cumplido su cometido y se eliminan. Siendo más preciso, el mensaje de control se encola en la QUEUE nº 0 y una copia de él se pasa al espacio de usuario para que procese la información de ruta, en este instante el paquete abandona la cola y se elimina. Esta manera de proceder esta implementado en la función *callback_external ( )* del archivo *ExternalQueu.c*, aun siendo un enfoque distinto, el código es idéntico al del punto 1.

4.1.5 Diagrama de funciones

A continuación se representa el diagrama de las funciones principales del plano de datos:

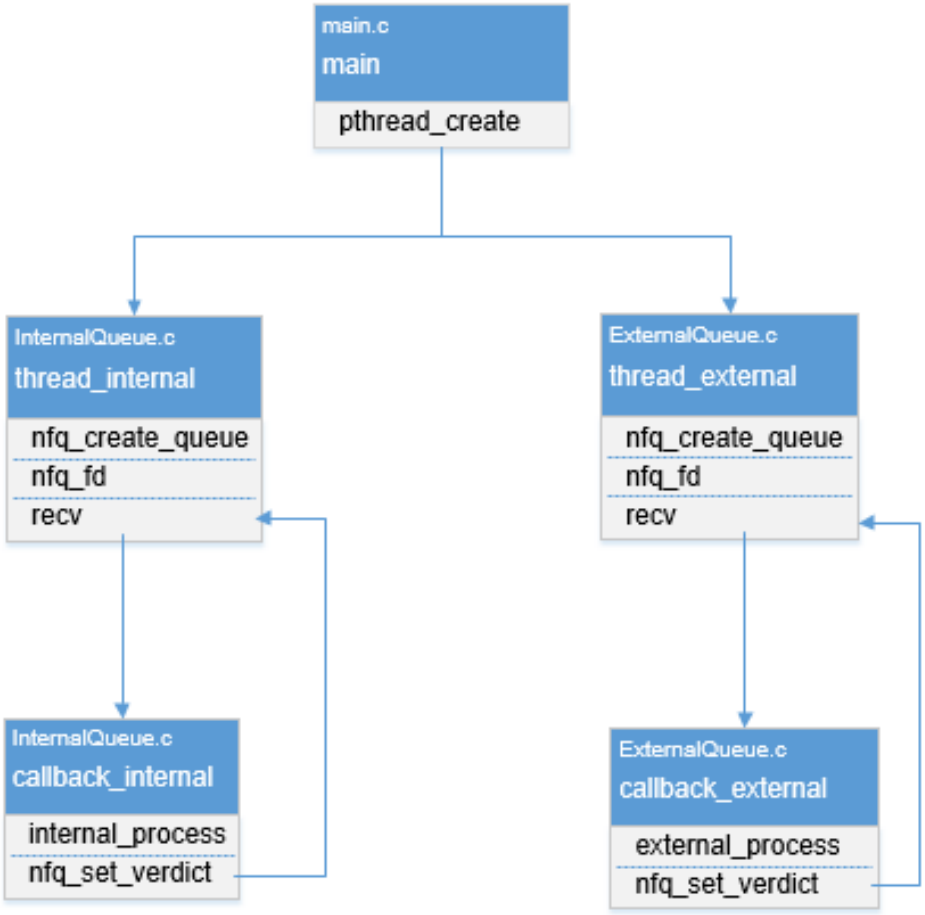


Figura 10.Diagrama de las funciones principales del plano de datos

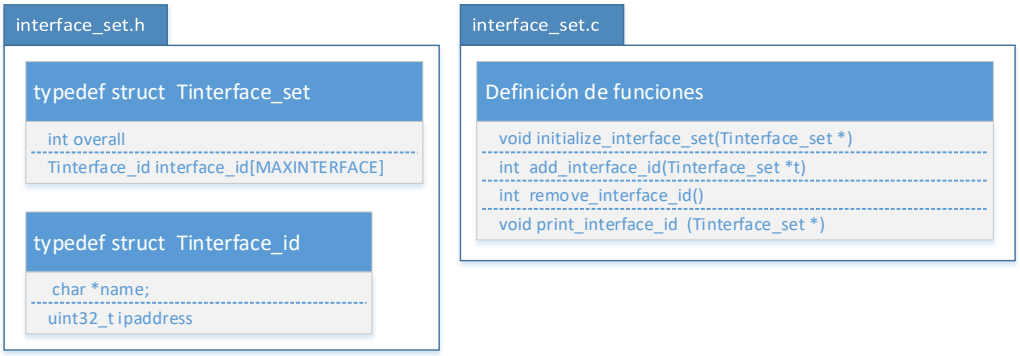
4.2 Plano de control

Tal como se citaba en la introducción del capítulo 4, el plano de datos engloban los procesos propios del protocolo de encaminamiento AODVv2. Todos los procesos que se han implementado fueron descritos al detalle en el capítulo 2 por lo que en este apartado se representaran los diagramas de funciones de cada uno de los procesos, y en el caso de las estructuras, su definición.

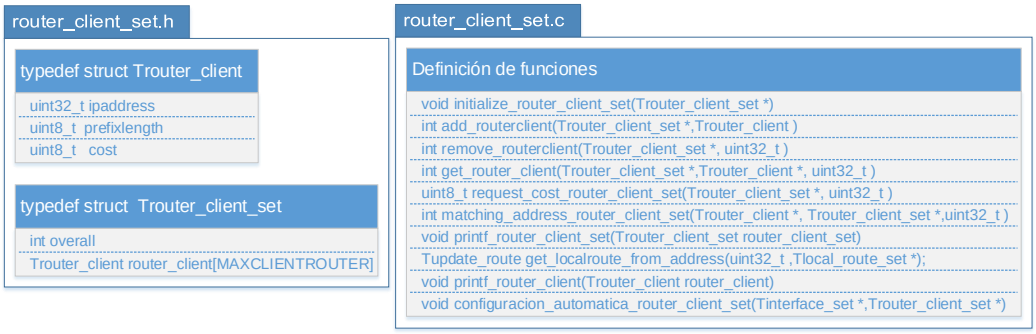
4.2.1 Estructura de datos

En este apartado se detallaran las estructuras de datos que se han utilizado para la implementación.

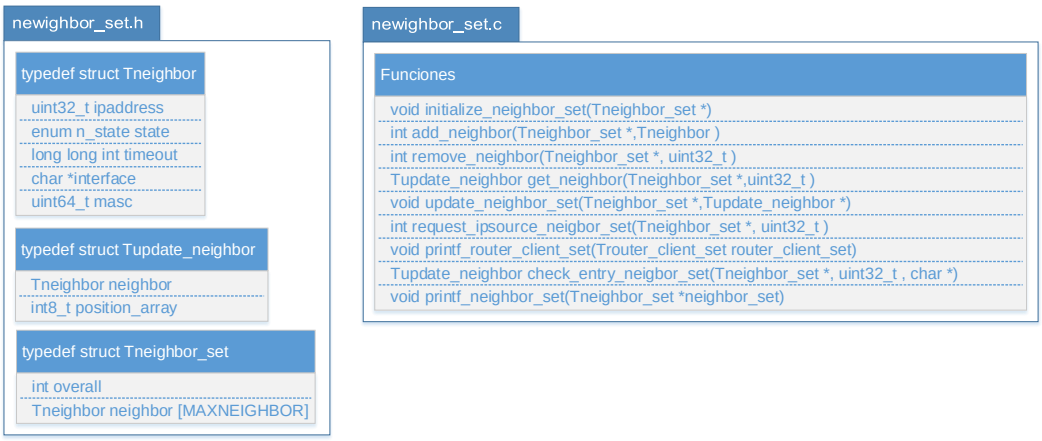
4.2.1.1 Interfaceset



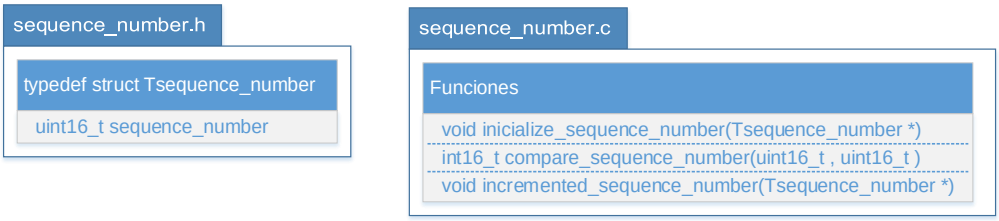
4.2.1.2 Router Client Set



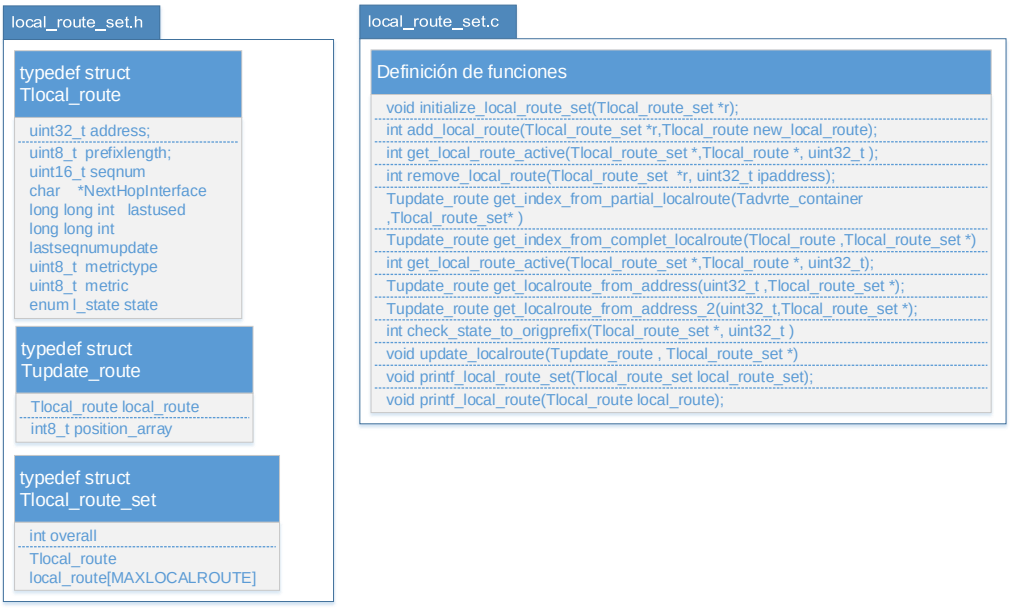
4.2.1.3 Neighbor Set



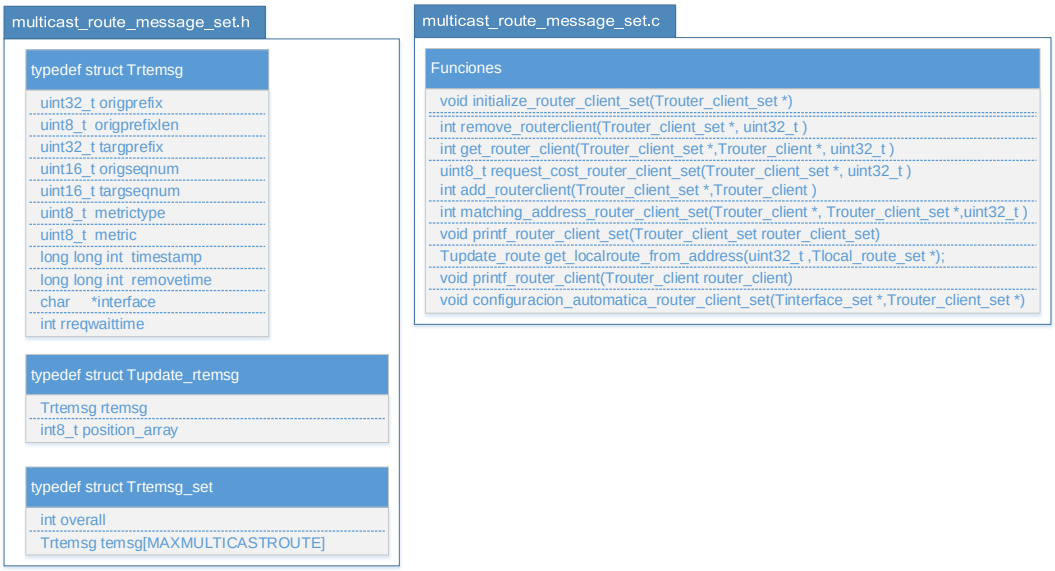
4.2.1.4 Sequence Number



4.2.1.5 Local Route Set



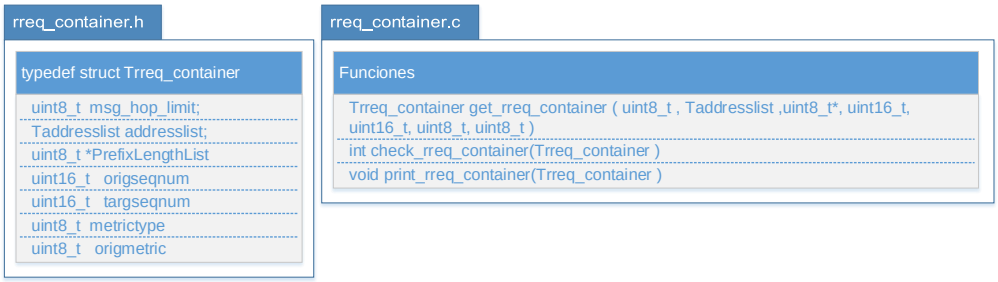
4.2.1.6 Multicast Route Message Set



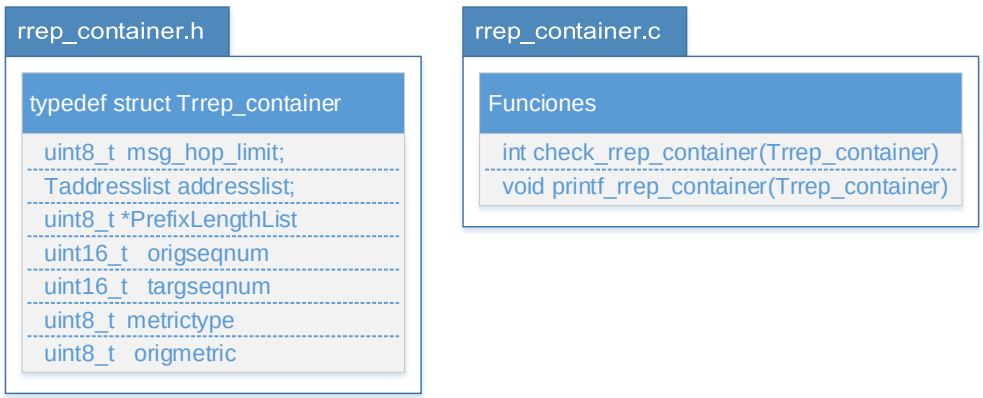
4.2.2 Mensajes

En este apartado se detallan las estructuras de datos que se han utilizado para implementar los mensajes de control.

4.2.2.1 Route Request (RREQ)



4.2.2.2 Route Reply (RREP)



4.2.2.3 Route Reply Acknowledgement

Este tipo de mensaje no necesita de una estructura de datos, ya que no transporta información.

4.2.3 Procesos

En este apartado se detallan las funciones que se han utilizado para implementar los diferentes procesos del protocolo AODVv2, cada una de ellas tiene una breve descripción de funcionamiento.

4.2.3.1 Neighbor Set Update

Para implementar este proceso se utilizan dos funciones definidas en el archivo *neighbor_set_update_process.h.*

- *int receipt_RREQ_neighbor_set(Tdata_pointers *, Tdata_packet *)*

  Esta función realiza una consulta a la tabla *Neighbor Set* si la dirección IP que incluye la estructura *Tdata_packet* existe:

  - *Tupdate_neighbor get_neighbor(Tneighbor_set *,uint32_t)*

  Sí no existe añade una nueva entrada en la tabla *Neighbor Set:*

  - *int add_neighbor(Tneighbor_set *, Tneighbor)*

- int receipt_RREP_update_neighbor_set(Tdata_pointers *,Tdata_packet *)

  Esta función realiza una consulta a la tabla *Neighbor Set* si la dirección IP que incluye la estructura *Tdata_packet* existe.

  Si no existe añade una nueva entrada en la tabla *Neighbor Set,* y añade la ruta hacia la dirección IP en la tabla de encaminamiento RIB con la siguiente función:

  - *bool add_host_in_RIB(uint32_t, char *)*

  Si existe actualiza el campo STATE de la entrada de la *neighbor set* la entrada confirmando que el enlace con el host es bidireccional:

  - void update_neighbor_set(Tneighbor_set *,Tupdate_neighbor *)

  Y añade la ruta hacia el HOST en la tabla de encaminamiento RIB.

4.2.3.2 Procesado de la información de ruta

Este proceso se implementa con dos funciones definidas en el archivo *received_route_information_process.h*:

- Tadvrte_container get_advrte_container_RREQ (Trreq_container, uint32_t, Tmetric_set *);

Esta función adapta la estructura Trreq_container a una Tadvrte_container, que será la que utilicen los procesos de evaluación y actualización de la ruta

- Tadvrte_container get_advrte_container_RREP (Trrep_container, uint32_t, Tmetric_set *)

Esta función adapta la estructura Trrep_container a una Tadvrte_container, que será la que utilicen los procesos de evaluación y actualización de la ruta.

4.2.3.4 Evaluación de la información de ruta

Este proceso se implementa con tres funciones definidas en el archivo *evaluating_route_information_process.h*, y tiene como objetivo determinar si la información de la ruta contenida en la estructura Tadvrte_container se utilizará para actualizar la tabla *Local Route Set*.

- *int evaluating_route_information (Tadvrte_container, Tlocal_route_set *, Tlocal_route_set *)*

  Es la función principal del proceso, y realiza la llamada a las dos funciones de apoyo.

- *Tlocal_route_set search_local_routes_step (Tadvrte_container, Tlocal_route_set *)*

  Está función se encarga de realizar la búsqueda en la tabla *Local Route Set.*

- *int compare_fields (Tlocal_route_set *,Tadvrte_container *)*

  Está función se encarga de comparar el coste y el número de secuencia del Tadvrte_container con la entrada correspondiente de la *Local Route Set.*

4.2.3.5 Actualización de la información de ruta

Este proceso se implementa con las funciones definidas en el archivo *applying_route_updates_process.h*. El código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles.

- *int update_route_process (Tdata_pointers *, char *, Tadvrte_container, Tlocal_route_set *);*

  Función principal del proceso, se encarga de realizar las llamadas al resto de funciones de apoyo, y las tareas del paso 1 del proceso

- *Tupdate_route update_local_route_step2 (Tdata_pointers *, Tlocal_route_set *, Tadvrte_container, char*)*

  Implementación del paso 2 del proceso.

- *int update_local_route_step3 (Tdata_pointers *, char*,Tlocal_route, Tadvrte_container)*

  Implementación del paso 3 del proceso.

- *int add_new_local_route_step4(Tdata_pointers * ,Tlocal_route *, Tadvrte_container);*

Implementación del paso 4 del proceso.

- *Tupdate_route update_partial_route_step5 (Tdata_pointers *, char*, Tlocal_route, Tadvrte_container, unsigned)*

Implementación del paso 5 del proceso.

- *int update_state_route_step6(Tdata_pointers * ,Tadvrte_container ,Tupdate_route)*

  Implementación del paso 6 del proceso

4.2.3.6 Eliminación de los mensajes de ruta redundantes usando la Multicast Route Message Set

Este proceso está implementado con las funciones definidas en el archivo *suppressing_redundant_messages_process.h*.

- *int suppressing_redundant_messages_process (Tdata_pointers *, Trreq_container *,Trrep_container *,char *);*

  Es la función principal y el código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles.

- *int create_new_rtemsg (Tdata_pointers *,uint32_t, uint8_t, uint32_t, uint16_t , uint16_t , char *,uint8_t, uint8_t);*

  Función de apoyo que agrega una nueva entrada de la tabla *Multicast Route Message Set*.

- *void update_complet_rtemsg (Tdata_pointers *,Tupdate_rtemsg, uint16_t, int16_t, uint8_t, Trreq_container *,Trrep_container *)*

  Función de apoyo que actualiza una entrada de la tabla *Multicast Route Message Set*.

- *void update_partial_rtemsg (Tdata_pointers *,Tupdate_rtemsg)*

  Función de apoyo que actualiza una entrada de la tabla *Multicast Route Message Set*.

4.2.3.7 Creación de mensajes RREQ

Este esta implementado por una función definida en el archivo rreq_generation_process.h.

- *Trreq_container process_rreq_generation (Tnetwork_prefix, Tnetwork_prefix, Trouter_client *, Tdata_pointers *)*

  Está función genera una estructura Trreq_container, los campos se asignan según lo expuesto en el apartado 2.3.1. Se apoya en dos funciones:

- *void incremented_sequence_number (Tsequence_number *)*

  Incrementa en 1 el número de secuencia del router.

- *uint16_t request_targaddr_local_route_set (uint32_t, Tlocal_route_set *)*

  Realiza una consulta a la tabla *Local Route Set* para buscar una entrada no valida que contenga el número de secuencia del router de destino.

4.2.3.8 Recepción de mensajes RREQ

Este proceso esta implementado por una función definida en el archivo rreq_reception_process.h.

- *int process_rreq_reception (Tdata_pointers *,Tadvrte_container *, Tdata_packet *, Trreq_container)*

Realiza las operaciones a realizar cuando un router AODVv2 recibe un mensaje RREQ, el código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles. Se apoya en las siguientes funciones:

- *int receipt_RREQ_neighbor_set (Tdata_pointers *, Tdata_packet *)*

  Comentada en el apartado 4.2.3.1.

- *int check_rreq_container (Trreq_container)*

  Comprueba que los campos del mensaje RREQ no sean nulos y chequea que las direcciones de OrigPrefix y TargPrefix sean válidas.

- *int check_metrictype (Tmetric_set *,uint8_t)*

  Comprueba que el tipo de métrica este soportado por el router AODVv2

- *int check_maximum_metric_value (uint8_t,Tmetric_set *)*

  Comprueba que el coste de la ruta del RREQ no exceda del máximo permitido para el tipo de métrica.

- *Tadvrte_container get_advrte_container_RREQ (Trreq_container, uint32_t, Tmetric_set *)*

Comentada en el apartado 4.2.3.2.

- *int evaluating_route_information (Tadvrte_container, Tlocal_route_set *, Tlocal_route_set *)*

  Comentada en el apartado 4.2.3.4.

- *int update_route_process (Tdata_pointers *,char *,Tadvrte_container, Tlocal_route_set *)*

  Comentada en el apartado 4.2.3.5.

- *int suppressing_redundant_messages_process(Tdata_pointers *, Trreq_container *,Trrep_container *, char *)*

  Comentada en el apartado 4.2.3.6.

- *int matching_address_router_client_set(Trouter_client *, Trouter_client_set *,uint32_t)*

  Compara el campo TargPrefix del RREQ con las entradas de la tabla *Router Client Set,* si hay coincidencia genera un mensaje RREP, si no reenvía el mensaje RREQ

4.2.3.9 Reenvío de mensajes RREQ

Este proceso esta implementado por una función definida en el archivo rreq_forwarding_process.h.

- *int rreq_forwarding_process (Trreq_container *, Tadvrte_container *,Tdata_pointers *)*

  Se encarga de realizar las operaciones para el reenvío de los mensajes RREQ. El código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles.

4.2.3.10 Creación de mensajes RREP

Este proceso esta implementado por una función definida en el archivo rrep_generation_process.h.

- *int rrep_generation_process(Trreq_container *, Tdata_pointers*, Trrep_container *)*

  Se encarga de realizar las operaciones para el reenvío de los mensajes RREQ. El código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles.

- *int get_router_client (Trouter_client_set *,Trouter_client *, uint32_t)*

  Devuelve el puntero de la entrada de la tabla Router Client Set que coincida con una dirección IP

- *void incremented_sequence_number (Tsequence_number *)*

  Comentada en el apartado 4.2.3.7

- *uint8_t request_cost_router_client_set (Trouter_client_set *, uint32_t)*

  Obtiene la métrica para el campo TargMetric del mensaje RREP

4.2.3.11 Recepción de mensajes RREP

Este proceso esta implementado por una función definida en el archivo rreq_reception_process.h.

- *int rrep_reception_process(Tdata_pointers *, Tdata_packet *,Trrep_container, Tadvrte_container *)*

Realiza las operaciones que se llevan a cabo cuando un router AODVv2 recibe un mensaje RREP, el código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles. Se apoya en las siguientes funciones:

- *int check_rreq_container (Trreq_container)*

  Comprueba que los campos del mensaje RREQ no sean nulos y chequea que las direcciones de OrigPrefix y TargPrefix sean válidas.

- *int check_metrictype (Tmetric_set *,uint8_t)*

  Comprueba que el tipo de métrica este soportado por el router AODVv2

- *int request_entry_multiple_fields(Trtemsg_set *,Trrep_container, char*,int);*

  Comprueba si el mensaje RREP era esperado o no

- *int receipt_RREP_update_neighbor_set (Tdata_pointers *,Tdata_packet *);*

Comentado en el apartado 4.2.3.1

- *Tadvrte_container get_advrte_container_RREP (Trrep_container, uint32_t, Tmetric_set *)*

  Comentada en el apartado 4.2.3.2.

- *int evaluating_route_information (Tadvrte_container, Tlocal_route_set *, Tlocal_route_set *)*

  Comentada en el apartado 4.2.3.4.

- *int update_route_process (Tdata_pointers *,char *,Tadvrte_container, Tlocal_route_set *)*

  Comentada en el apartado 4.2.3.5.

- *int suppressing_redundant_messages_process (Tdata_pointers *, Trreq_container *,Trrep_container *,char *);*

Comentada en el apartado 4.2.3.6.

- *int check_state_to_origprefix (Tlocal_route_set *, uint32_t)*

  Realiza una consulta a la tabla Local Route Set para comprobar si el estado de la ruta es válida o no está confirmada.

4.2.3.12 Reenvío de mensajes RREP

Este proceso esta implementado por una función definida en el archivo rrep_forwarding_process.h.

- *int rrep_forwarding_process (Trrep_container *, Tadvrte_container *,Tdata_pointers *)*

Se encarga de realizar las operaciones para el reenvío de los mensajes RREP. El código está estructurado siguiendo los mismos pasos que lo expuesto en el draft, consultar el anexo A para más detalles.

4.3 RFC5444 [3]

En este apartado se detalla la implementación del paquete definido en el RFC5444 [3]. El código está dividido en dos bloques:

- El primero se encarga de adaptar un mensaje de control con el formato definido en el apartado 2.3 en un *array* de bytes con el formato definido en el RFC5444 [3]. Para ello se utiliza una serializado simple.
- El segundo bloque se encarga de la operación contraria, a partir de un array de bytes estructurado según el RFC5444 [3] obtener un mensaje control. Para esta operación se usa una técnica llamada *Parcing*.

En los siguientes apartados se detalla cada uno de los bloques y se muestra el resultado mediante un ejemplo real.

4.3.1 Serializado de los mensajes de control

Serializar consiste en codificar de una estructura de datos con el fin de transmitirlo por una interfaz de red. Para dar un ejemplo de cada fase supondremos que partimos del siguiente un mensaje de control que es el resultado de ejecutar una función que imprime por consola el contenido de un mensaje RREQ:

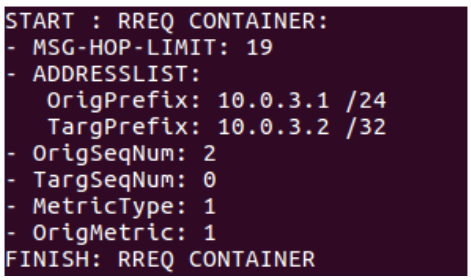


Figura 11. Captura pantalla contenido mensaje RREQ

Este proceso se realiza en dos fases:

1. La primera fase se encarga de convertir un mensaje de control que tiene una estructura como las definidas en el apartado 2.3, en una estructura de datos que sigue el formato RFC5444 [3].

Para ello se utilizan las siguientes funciones:

- *Pgman_packet get_packet_rreq (Trreq_container, Tpointer_gman_packet *)*

  Está función se encarga de adaptar los mensajes RREQ al formato RFC5444 [3]. En la figura 12 se muestra el resultado de la adaptación:

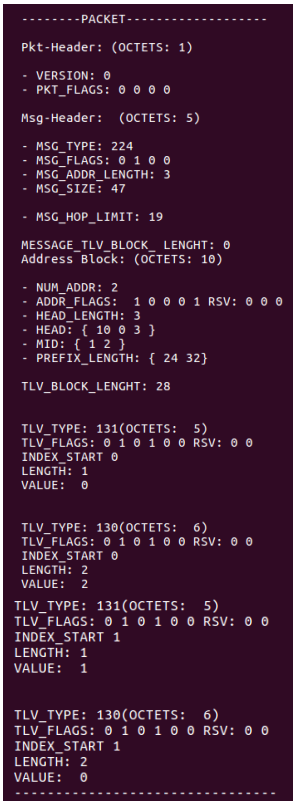


Figura 12. Captura de pantalla de la estructura de un paquete RFC5444 que contiene el mensaje RREQ

- *Pgman_packet get_packet_rrep (Trrep_container, Tpointer_gman_packet *)*

  Está función se encarga de adaptar los mensajes RREP al formato RFC5444 [3].

- *Pgman_packet get_packet_rrep_ack (uint8_t, Tpointer_gman_packet *)*

  Está función se encarga de adaptar los mensajes RREP_Ack con y sin el AckReq al formato RFC5444 [3].

2. La segunda fase se encarga de convertir una estructura de datos que sigue el formato RFC5444 [3] en el array de bytes que se transmitirá por la interfaz. Para ello se utiliza la siguiente función:

   - *uint8_t* get_array_packet (Pgman_packet, uint8_t * ,Tpointer_gman_packet *)*

La siguiente figura 13 se muestra la captura de pantalla de una función que imprime por pantalla el contenido del array de bytes.

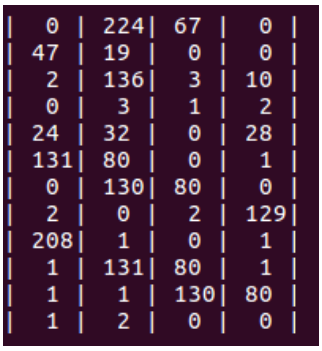


Figura 13. Contenido de array de bytes

4.3.2 *Parcing*

Parcing se le llama a la técnica que consiste en analizar una secuencia de símbolos a fin de determinar su estructura de datos con respecto a una estructura formal dada.

La idea es obtener a partir de una array de bytes contenido en un buffer de datos resultado de un paquete de datos recibidos en el mensaje de control correspondiente. Para ello se usan las siguientes funciones, que no se detallaran porque son el resultado inverso al obtenido en el apartado 4.3.1

- Pgman_packet get_parse_array (uint8_t *, Tpointer_gman_packet_parse *)
- Trreq_container get_parse_container_rreq (Pgman_packet, Tpointer_gman_packet_parse *)
- Trrep_container get_parse_container_rreq (Pgman_packet, Tpointer_gman_packet_parse *)
- uint8_t get_parse_container_rrep_ack (Pgman_packet)

CAPÍTULO 5. ESCENARIO EXPERIMENTAL

En este capítulo se detalla el proceso que se ha seguido para poner a prueba la implementación del protocolo AODVv2. Paso a paso se detallan los pormenores que han ido sucediendo a lo largo de la fase experimental. A continuación se muestra la topología del escenario experimental:

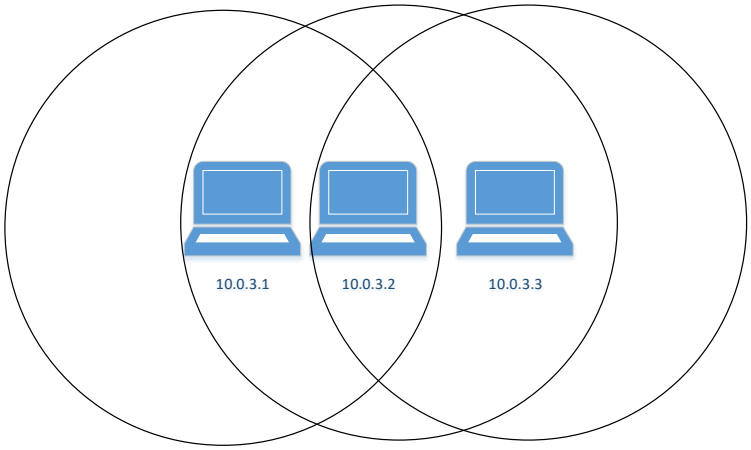


Figura 14. Topología del escenario experimental

- Los nodos serán tres ordenadores portátiles con las siguientes características:

  - CPU 64 bits
  - Interface radio 802.11n
  - Ubuntu 16.4.3 LTS 64 bits

- Las direcciones IP de las interfaces pertenecen a la misma red:

  - P1: 10.0.3.1/24
  - P2: 10.0.3.2/24
  - P3: 10.0.3.3/24

- Las tres máquinas se conectan a la misma red Ad Hoc

5.1 Simulación de las coberturas

El espacio donde se han realizado las pruebas no ha permitido montar un escenario real donde pudiéramos tener control sobre los enlaces radio de la red.

Para solucionar este problema, se ha utilizado una regla específica para cada máquina que permitió descartar todo el tráfico a nivel 3 procedente de un nodo en concreto. Estas reglas se incluyeron en el script *Confnetfilter.sh* de cada máquina y son las siguientes:

- P1: iptables –t mangle –A PREROUTING –m mac -–mac-source 48:5d:60:7e:bb:8f –j DROP

Todo el tráfico procedente de la MAC del P3 se bloquea.

- P3: iptables –t mangle –A PREROUTING –m mac -–mac-source C0:25:e9:24:de:4c –j DROP

  Todo el tráfico procedente de la MAC del P1 se bloquea.

5.2 Tráfico superpuesto

Una vez se ejecutaba el programa en la máquina, procesos internos del sistema generaban tráfico, como el que cumple los requisitos impuestos por la regla:

- *-A OUTPUT -j NFQUEUE --queue-num 1*

Netfilter lo envía a la cola nº 1 y la función de *callback* se ejecuta, pero lanzaba un aviso por consola advirtiendo que la IP que estaba generando el flujo no coincidía con ninguna entrada de la tabla Router Client Set, por lo tanto descartaba el paquete.

Se agregó una rutina para que mostrara por pantalla información relativa al *hook-point* donde se había capturado el paquete, IP destino, IP origen e identificador de paquete. A continuación se muestra una captura de pantalla con todo el proceso:

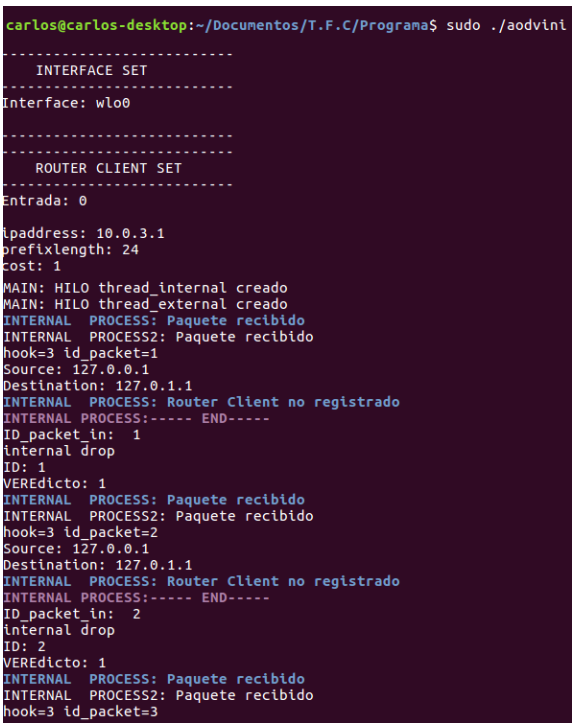


Figura 15 Captura de pantalla

Resulto que la IP en cuestión, era la dirección de *loopback* 127.0.0.1. Para evitar interferir sobre el funcionamiento del sistema se decidió incluir una regla en la cadena OUTPUT. Ésta acepta todo el tráfico que provenga de una dirección de *loopback* 127.0.0.0/8; de esta manera *Netfilter* no lo envía a la cola nº 1. Esta regla fue incluida en el archivo *Confnetfilter.sh* de cada máquina:

- *-I OUTPUT -s 127.0.0.1/8 -j ACCEPT*

5.3 IPv6

En esta fase del proyecto se decidió que la implementación del protocolo soportara solo direcciones de red IPv4. Por ello se desactivaron las direcciones IPv6 de las interfaces que se iban a configurar en el Interface Set. Para ello se agregó la siguiente línea en el archivo *Confnetfilter.sh:*

- *sh -c 'echo 1 > /proc/sys/net/ipv6/conf/wlo0/disable_ipv6'*

5.4 Descubrimiento de Ruta

La estrategia de implementación que se ha adoptado en este proyecto se basa en Netfilter y en una de sus características que permite enviar al espacio de usuario los paquetes que pasan por la cadena OUTPUT.

Vamos a suponer que en la maquina P1 del escenario experimental se inicia el protocolo AODVv2, y por lo tanto la tabla de encaminamiento está vacía. La idea que se tenía de cómo funcionaría el descubrimiento de una ruta era el siguiente:

1. Un proceso local genera un paquete con la dirección IP de destino X.X.X.X.

2. El paquete pasa por la cadena OUTPUT y se envía a la cola nº 1, gracias a la regla citada en el apartado 4.1.1.

3. Se ejecuta la función de *callback*, la cual hace una consulta a la tabla de encaminamiento para saber si hay alguna ruta hacia el destino. Como la tabla de encaminamiento está vacía, inicia el descubrimiento de una ruta.

4. El paquete se queda en la cola hasta que se finaliza el proceso y se obtiene un camino hacia la IP de destino.

Al hacer las primeras pruebas se observó que, tras ejecutar el ping, este devolvía el mensaje:

- *“Connect : Network is Unreachable”*

En principio este es un comportamiento normal de PING, ya que la tabla de encaminamiento está vacía, ¿Pero entonces se cómo inicia el proceso de descubrimiento de una ruta si el paquete no pasa por OUTPUT?

La solución que se adopto es cargar una ruta por defecto cuando se iniciara el programa, que permitiera que el paquete entrará por la cadena OUTPUT y se encolara en QUEUE nº1. Se añadió la siguiente línea en el archivo *Confnetfilter.sh* de cada máquina:

- *route add -net 0 metric 1000 dev wlo0*

Con esta ruta por defecto ya era posible generar el descubrimiento para destinos que no estuvieran en la tabla de encaminamiento.

5.5 Comunicación entre dos nodos directamente conectados

En este apartado se muestran los resultados experimentales obtenidos del proceso de descubrimiento de una ruta entre dos máquinas que estaban directamente conectadas.
Se ejecutó un ping desde la maquina P3 contra la maquina P2 y el resultado fue el esperado: el proceso de descubrimiento de ruta para la dirección 10.0.3.2 se hizo correctamente A continuación se muestra la captura completa del proceso:

No.	Time	Source	Destination	Protocol	Lengt	Info
1	0.000000000	10.0.3.1	10.0.3.255	packetbb	90	
2	0.006367127	LiteonTe_f9:c0:1a	Broadcast	ARP	42	Who has 10.0.3.1? Tell 10.0.3.2
3	0.006397522	Tp-LinkT_24:dc:4c	LiteonTe_f9:c0:1a	ARP	42	10.0.3.1 is at c0:25:e9:24:dc:4c
4	0.010856055	10.0.3.2	10.0.3.1	packetbb	51	
5	0.011083915	10.0.3.1	10.0.3.2	packetbb	49	
6	0.011188997	10.0.3.2	10.0.3.1	packetbb	83	
7	0.019323606	10.0.3.1	10.0.3.2	ICMP	98	Echo (ping) request id=0x0b1c, seq=1/256, ttl=64 (reply in 8)
8	0.023822131	10.0.3.2	10.0.3.1	ICMP	98	Echo (ping) reply id=0x0b1c, seq=1/256, ttl=64 (request in 7)
9	1.000927699	10.0.3.1	10.0.3.2	ICMP	98	Echo (ping) request id=0x0b1c, seq=2/512, ttl=64 (reply in 10)
10	1.002611621	10.0.3.2	10.0.3.1	ICMP	98	Echo (ping) reply id=0x0b1c, seq=2/512, ttl=64 (request in 9)
11	2.002740020	10.0.3.1	10.0.3.2	ICMP	98	Echo (ping) request id=0x0b1c, seq=3/768, ttl=64 (reply in 12)
12	2.007126856	10.0.3.2	10.0.3.1	ICMP	98	Echo (ping) reply id=0x0b1c, seq=3/768, ttl=64 (request in 11)
13	3.004254157	10.0.3.1	10.0.3.2	ICMP	98	Echo (ping) request id=0x0b1c, seq=4/1024, ttl=64 (reply in 14)
14	3.006892224	10.0.3.2	10.0.3.1	ICMP	98	Echo (ping) reply id=0x0b1c, seq=4/1024, ttl=64 (request in 13)
15	4.006019410	10.0.3.1	10.0.3.2	ICMP	98	Echo (ping) request id=0x0b1c, seq=5/1280, ttl=64 (reply in 16)
16	4.007157814	10.0.3.2	10.0.3.1	ICMP	98	Echo (ping) reply id=0x0b1c, seq=5/1280, ttl=64 (request in 15)

Figura 16.Captura del proceso de descubrimiento de ruta

En el anexo B se puede consultar el detalle de los mensajes de control.

5.6 Comunicación entre dos nodos a 2 saltos de distancia

En este apartado se muestran los resultados experimentales obtenidos del proceso de descubrimiento de una ruta entre dos máquinas que estaban a 2 saltos de distancia.

Se montó el escenario de la figura 11, en este caso se añadieron en el archivo *Confnetfilter.sh* de P1 y P3 las reglas para simular la cobertura tal como se vio en el apartado 5.1.

Se ejecutó un ping entre la maquinas P1 y P3 y se obtuvo el siguiente resultado:

No.	Time	Source	Destination	Protocol	Lengt	Info
1	0.000000000				110	<Ignored>
2	82.079640622	10.0.3.1	10.0.3.255	packetbb	90	
3	82.080020345	10.0.3.2	10.0.3.255	packetbb	90	
4	82.083679980	Azurewav_7e:bb:8f	Broadcast	ARP	42	Who has 10.0.3.2? Tell 10.0.3.3
5	82.083700176	LiteonTe_f9:c0:1a	Azurewav_7e:bb:8f	ARP	42	10.0.3.2 is at d0:df:9a:f9:c0:1a
6	82.088365351	10.0.3.3	10.0.3.2	packetbb	51	
7	82.088578191	10.0.3.2	10.0.3.3	packetbb	49	
8	82.089640099	10.0.3.3	10.0.3.2	packetbb	83	
9	82.094675164	LiteonTe_f9:c0:1a	Broadcast	ARP	42	Who has 10.0.3.1? Tell 10.0.3.2
10	82.097166739	Tp-LinkT_24:dc:4c	LiteonTe_f9:c0:1a	ARP	42	10.0.3.1 is at c0:25:e9:24:dc:4c
11	82.097187750	10.0.3.2	10.0.3.1	packetbb	51	
12	82.097220966	10.0.3.2	10.0.3.1	packetbb	83	
13	82.099915525	10.0.3.1	10.0.3.2	packetbb	49	
14	83.112747771	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=2/512, ttl=64 (no response found!)
15	83.112782631	10.0.3.2	10.0.3.1	ICMP	126	Redirect (Redirect for host)
16	83.112815356	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=2/512, ttl=63 (reply in 17)
17	83.115787061	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=2/512, ttl=64 (request in 16)
18	83.115818154	10.0.3.2	10.0.3.3	ICMP	126	Redirect (Redirect for host)
19	83.115841549	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=2/512, ttl=63
20	84.112573817	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=3/768, ttl=64 (no response found!)
21	84.112619818	10.0.3.2	10.0.3.1	ICMP	126	Redirect (Redirect for host)
22	84.112651821	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=3/768, ttl=63 (reply in 23)
23	84.115495309	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=3/768, ttl=64 (request in 22)
24	84.115514056	10.0.3.2	10.0.3.3	ICMP	126	Redirect (Redirect for host)
25	84.115531641	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=3/768, ttl=63
26	85.113996670	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=4/1024, ttl=64 (no response found!)
27	85.114051667	10.0.3.2	10.0.3.1	ICMP	126	Redirect (Redirect for host)
28	85.114090065	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=4/1024, ttl=63 (reply in 29)
29	85.119029253	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=4/1024, ttl=64 (request in 28)
30	85.119053992	10.0.3.2	10.0.3.3	ICMP	126	Redirect (Redirect for host)
31	85.119080798	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1caa, seq=4/1024, ttl=63
32	86.115166078	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1caa, seq=5/1280, ttl=64 (no response found!)
33	86.115212467	10.0.3.2	10.0.3.1	ICMP	126	Redirect (Redirect for host)

Figura 17 Captura del tráfico desde la maquina P2

Una vez se finalizaba el descubrimiento de la ruta, la maquina P2 contestaba a los *ICMP Request* que le envía P1 con un *ICMP redirect.* Esto provoca que la inmensa mayoría de ICMP Request no llegue a P3 y por lo tanto se pierdan paquetes.

Después de analizar el problema se llegó a la siguiente conclusión:

El *ICMP redirect* intenta informar a los nodos de la red para que actualicen las rutas de acuerdo a lo que los distintos nodos han considerado como óptimo (desde el punto de vista de IP únicamente). Como se está empleando un protocolo de encaminamiento para construir dichas rutas, esto genera un conflicto, puesto que el concepto de "optimo" lo impone AODVv2. Por lo tanto, lo adecuado en este caso es desactivar los mensajes *ICMP redirect*.

Los *ICMP redirect* pueden ser válidos en redes cableadas, donde no existe el problema del nodo oculto. Sin embargo, dicho problema está presente en redes inalámbricas como las que consideramos, por lo que deben ser filtrados.

Para desactivar el ICMP redirect se añadieron los siguientes comandos en el archivo *Confnetfilter.sh* de cada máquina:

- #echo 0 > /proc/sys/net/ipv4/conf/all/send_redirect
- #echo 0 > /proc/sys/net/ipv4/conf/wlo0/send_redirect

Se puso a prueba de nuevo el escenario y desde la consola de P1 se observó lo siguiente:

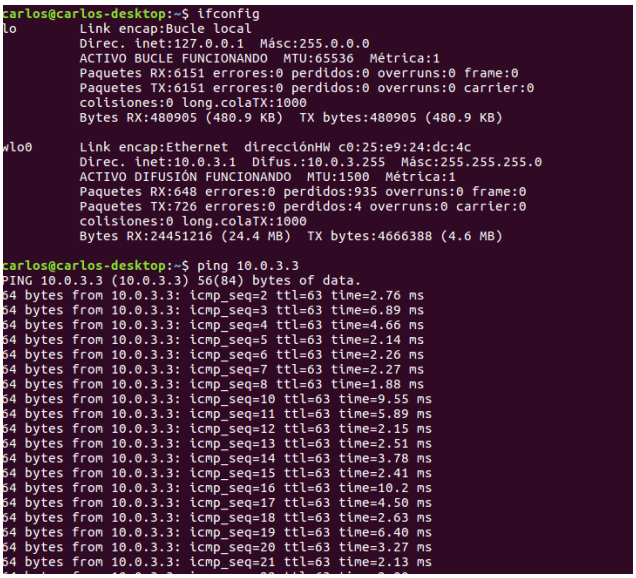


Figura 18.Comando ping desde P1 a P3

A simple vista funciona correctamente, pero al repetir el experimento se produjo el mismo resultado una y otra vez. El primer paquete del flujo se pierde.

No se ha podido determinar el motivo, quedará pendiente para trabajos futuros resolver este problema.

Comprobamos que las rutas se han añadido correctamente en la Tabla de encaminamiento de P1, P2 y P3:

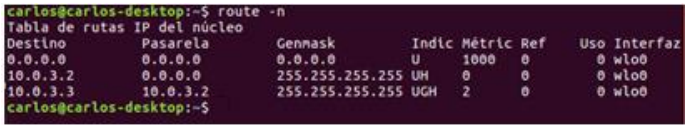


Figura 19 Tabla de encaminamiento de P1

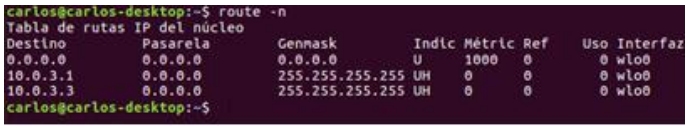


Figura 20. Tabla de encaminamiento de P2

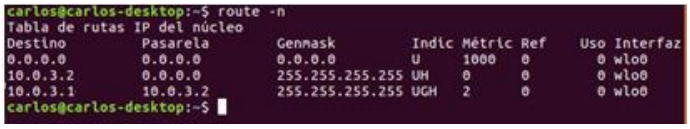
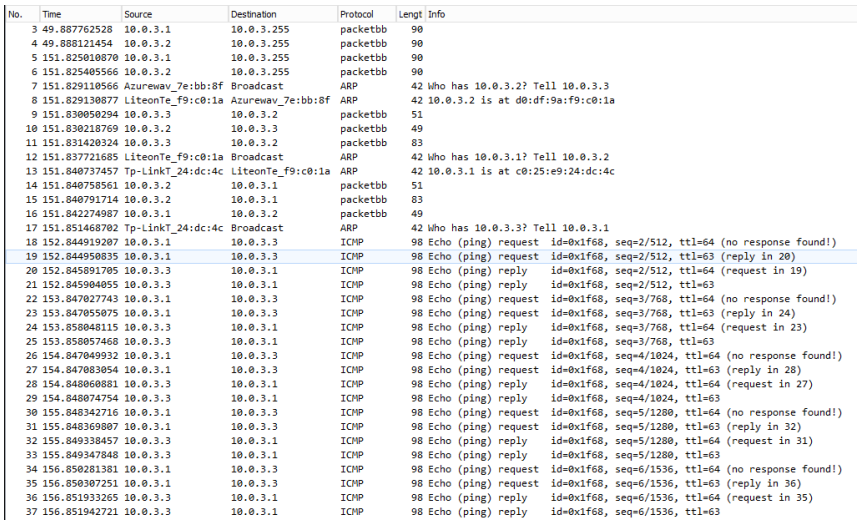


Figura 21. Tabla de encaminamiento de P3

Y por último la captura de tráfico:



No.	Time	Source	Destination	Protocol	Lengt	Info
3	49.887762528	10.0.3.1	10.0.3.255	packetbb	90	
4	49.888121454	10.0.3.2	10.0.3.255	packetbb	90	
5	151.825010870	10.0.3.1	10.0.3.255	packetbb	90	
6	151.825405566	10.0.3.2	10.0.3.255	packetbb	90	
7	151.829110566	Azurewav_7e:bb:8f	Broadcast	ARP	42	Who has 10.0.3.2? Tell 10.0.3.3
8	151.829130877	LiteonTe_f9:c0:1a	Azurewav_7e:bb:8f	ARP	42	10.0.3.2 is at d0:df:9a:f9:c0:1a
9	151.830050294	10.0.3.3	10.0.3.2	packetbb	51	
10	151.830218769	10.0.3.2	10.0.3.3	packetbb	49	
11	151.831420324	10.0.3.3	10.0.3.2	packetbb	83	
12	151.837721685	LiteonTe_f9:c0:1a	Broadcast	ARP	42	Who has 10.0.3.1? Tell 10.0.3.2
13	151.840737457	Tp-LinkT_24:dc:4c	LiteonTe_f9:c0:1a	ARP	42	10.0.3.1 is at c0:25:e9:24:dc:4c
14	151.840758561	10.0.3.2	10.0.3.1	packetbb	51	
15	151.840791714	10.0.3.2	10.0.3.1	packetbb	83	
16	151.842274987	10.0.3.1	10.0.3.2	packetbb	49	
17	151.851468702	Tp-LinkT_24:dc:4c	Broadcast	ARP	42	Who has 10.0.3.3? Tell 10.0.3.1
18	152.844919207	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=2/512, ttl=64 (no response found!)
19	152.844950835	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=2/512, ttl=63 (reply in 20)
20	152.845891705	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=2/512, ttl=64 (request in 19)
21	152.845904055	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=2/512, ttl=63
22	153.847027743	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=3/768, ttl=64 (no response found!)
23	153.847055075	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=3/768, ttl=63 (reply in 24)
24	153.858048115	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=3/768, ttl=64 (request in 23)
25	153.858057468	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=3/768, ttl=63
26	154.847049932	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=4/1024, ttl=64 (no response found!)
27	154.847083054	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=4/1024, ttl=63 (reply in 28)
28	154.848060881	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=4/1024, ttl=64 (request in 27)
29	154.848074754	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=4/1024, ttl=63
30	155.848342716	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=5/1280, ttl=64 (no response found!)
31	155.848369807	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=5/1280, ttl=63 (reply in 32)
32	155.849338457	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=5/1280, ttl=64 (request in 31)
33	155.849347848	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=5/1280, ttl=63
34	156.850281381	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=6/1536, ttl=64 (no response found!)
35	156.850307251	10.0.3.1	10.0.3.3	ICMP	98	Echo (ping) request id=0x1f68, seq=6/1536, ttl=63 (reply in 36)
36	156.851933265	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=6/1536, ttl=64 (request in 35)
37	156.851942721	10.0.3.3	10.0.3.1	ICMP	98	Echo (ping) reply id=0x1f68, seq=6/1536, ttl=63

Figura 22. Captura de tráfico desde P2

Como se puede observar una vez se finaliza el descubrimiento de la ruta, se empiezan a trasmitir los *ICMP Request* e *ICMP Reply.* El mensaje "*no response found*" que muestra el analizador de tráfico no lo logro comprender, ya que si analizamos los paquetes de una trasmisión completa, el proceso es correcto:

- P1 envía el *ICMP Request* con la dirección MAC destino P2 y con IP destino P3

```
> Frame 18: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: Tp-LinkT_24:dc:4c (c0:25:e9:24:dc:4c), Dst: LiteonTe_f9:c0:1a (d0:df:9a:f9:c0:1a)
> Internet Protocol Version 4, Src: 10.0.3.1, Dst: 10.0.3.3
> Internet Control Message Protocol
```

Figura 23. ICMP Request

- P2 reenvía el *ICMP Request* pero con la MAC destino P3, MAC origen P2 y IP destino P3

```
> Frame 19: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: LiteonTe_f9:c0:1a (d0:df:9a:f9:c0:1a), Dst: Azurewav_7e:bb:8f (48:5d:60:7e:bb:8f)
> Internet Protocol Version 4, Src: 10.0.3.1, Dst: 10.0.3.3
> Internet Control Message Protocol
```

Figura 24. ICMP Request

- P3 contesta al *ICMP Request* con un *ICMP Reply* con la MAC destino P2 y dirección IP destino P1

```
> Frame 20: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: Azurewav_7e:bb:8f (48:5d:60:7e:bb:8f), Dst: LiteonTe_f9:c0:1a (d0:df:9a:f9:c0:1a)
> Internet Protocol Version 4, Src: 10.0.3.3, Dst: 10.0.3.1
> Internet Control Message Protocol
```

Figura 25. ICMP Reply

- P2 reenvía el *ICMP Reply* pero con la MAC destino P1, MAC origen P2 y dirección IP destino P1

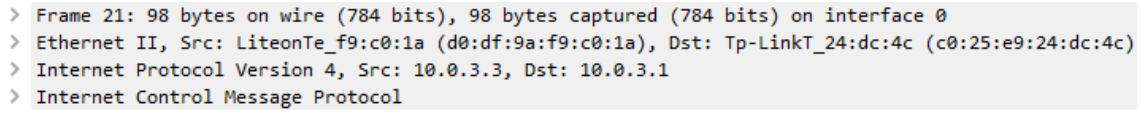


Figura 26. ICMP Reply

CAPITULO 6. CONCLUSIONES

No es posible entrar a valorar el protocolo AODVv2 porque no se ha finalizado su implementación. Pero lo que sí que puedo valorar son mis experiencias a lo largo de todo el proceso.

La curva de aprendizaje ha sido un factor clave que ha propiciado los retrasos en los tiempos previstos. Muchos conceptos nuevos a nivel de programación se han tenido que aplicar para llegar a implementar el proceso de descubrimiento de una ruta. Punteros, memoria dinámica, temporizadores, sincronización de thread's, operadores bit a bit, etc...

Yo partía con unos conocimientos previos, a mí parecer aceptables, pero me he dado cuenta que lo que se me exigía para enfrentarme a este proyecto me superaba. Solo sólo con muchísima dedición y aprendizaje autónomo por mi parte, y con la ayuda de los directores de este proyecto en los peores momentos, se ha llegado hasta el final.

En lo referente a la comprensión teórica del protocolo AODVv2 la parte más complicada ha sido entender el RFC5444, un galimatías ingenioso que permite universalizar el formato de los paquetes en las redes MANET.

A pesar de todos y cada uno de los obstáculos encontrados a lo largo del camino, el objetivo inicial del proyecto se ha alcanzado. Se ha podido implementar el protocolo AODVv2 en código C en casi su totalidad, y se ha podido ejecutar en una máquina con sistema operativo LINUX.

Como trabajo futuro se debería finalizar las implementaciones de los procesos que en este proyecto no se han podido abordar, como son el envío, recepción y renvío de mensajes de error de ruta (RRER). Esto permitiría el mantenimiento de la tabla de encaminamiento en caso de caída de los enlaces.

Tras haber completado estas tareas, el proyecto podrá seguir adelante con la fase número dos, la cual contempla la migración del código para su correcto funcionamiento en dispositivos como Raspberry Pi.

BIBLIOGRAFÍA

[1][Draft-ietf-manet-aodvv2-16] C. Perkins, S. Ratliff, J. Dowdell, L. Steenbrink V. Mercieca, “Ad Hoc On demand Distance Vector Version 2 (AODVv2) Routing”, Mayo 2016, <https://tools.ietf.org/html/draft-ietf-manet-aodvv2-16 >

[2] [RFC5498] Chakeres, I., "IANA Allocations for Mobile Ad Hoc Network (MANET) Protocols", RFC 5498, Marzo 2009. <https://tools. ietf.org/html /rfc5498>

[3] [RFC5444] Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, DOI 10.17487/RFC5444, Febrero 2009, <http://www.rfc-editor.org/info/rfc5444>

[4] Ian D. Chakeres, Elizabeth M. Belding,” Implementing and evaluating the DYMO routing protocol”, Mayo 2009,

[5][Draft-perkins-manet-aodvv2pkts-00] C. Perkins Futurewei, “ AODVv2 Example RFC 5444 Packets”, Marzo 2015. < https://tools.ietf.org/id/draft-perkins-manet-aodvv2pkts-00.html >

[6] Addison Wesley, “UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API “ISBN: 0-13-141155-1, Noviembre 2003.

[7] < http://www.netfilter.org >

ANEXO A. Detalles de los procesos y estructuras de datos

En este anexo se detallan varios de los procesos que intervienen en el protocolo AODVv2. La información que contiene este anexo es la misma que se pude consultar en el draft de AODVv2, pero como es la base mediante la cual se ha construido todo el código me ha parecido conveniente incluirlo.

Interface set

Esta clase solo tiene un campo:

- interface.id: Identificador único para cada elemento del interfaceset que permita a AODVv2 identificar exactamente una interfaz inalámbrica en un nodo. Se usará la dirección MAC del interfaz como identificador.

Numero de secuencia

El *Sequence Number* se formará como un número entero de 16 bit sin signo.

Cada router AODVv2 debe asegurarse de que su Sequence Number se incrementa en uno cuando se crea un RREQ o RREP, excepto cuando el número de secuencia es 65.535, en cuyo caso debe ser reajustado a uno. El valor cero se reserva para indicar que el *Sequence Number* es desconocido.

Un router AODVv2 sólo debe adjuntar su propio *Sequence Number* cuando alguno de los clientes registrados en la *Router Client Set* solicite información acerca de una ruta.

Todos los mensajes de ruta reenviados por otros routers deberán contener el Sequence Number del router que origino el mensaje.

Para determinar si la información recién recibida es obsoleta y por lo tanto redundante, el *Sequence Number* asociado a la información se compara con el *Sequence Number* de la información existente sobre la misma ruta. La comparación se lleva a cabo restando el *Sequence Number* existente del *Sequence Number* recibido. El resultado de la resta debe interpretarse como un entero de 16-bit con signo.

- Si el resultado es negativo, la información recibida se considera más antigua que la información existente, por lo que se descartada.

- Si el resultado es positivo, la información recién recibida se considera más reciente que la información existente, por lo que se procesa.

- Si el resultado es cero, la información recién recibida no se considera obsoleta, y por lo tanto se procesado para comprobar si es redundante.

Neighbor Set

Cada entrada debe contener de la tabla Neighbor Set debe contener:

- Neighbor.IPAddres: La dirección IP del *router* vecino, esta se aprende a partir de los mensajes RREP.
- Neighbor.State: Indica si el enlace con el encaminador vecino es bidireccional. Este campo puede solo tiene tres estados:
  - *Heard*: Es el estado inicial.
  - *Confirmed*: El enlace con el encaminador vecino es bidireccional.
  - *Blacklisted*: El enlace con el encaminador vecino es unidireccional.
- Neighbor.Timeout: Indica en qué momento se tiene que actualizar el Neighbor.State.
- Neighbor.Interface: La interfaz mediante la cual se establece el enlace con el vecino.

Router Client Set

Cada entrada del “Router Client Set” tiene que contener los siguientes campos:

- RouterClient.IPAddress: Una dirección IP o el inicio del intervalo de direcciones del cliente que precisa del servicio de descubrimiento de rutas.
- RouterClient.PrefixLength: La longitud (bits) del prefijo de encaminamiento asociado con el RouterClient.IPAddress. Si la longitud del prefijo no es igual a la longitud de la dirección de RouterClient.IPAddress, el encaminador AODVv2 debe participar en el descubrimiento de rutas en nombre de todas las direcciones dentro de ese prefijo.
- RouterClient.Cost: El coste asociado al cliente.

Multicast Route Message Set

Cada entrada de la tabla *multicast route message set* debe contener los siguientes campos:

- RteMsg.OrigPrefix: El prefijo asociado con el campo OrigAddr, que contiene la dirección de origen del paquete IP que ha activado la solicitud de ruta.

RteMsg.OrigPrefixLen: La longitud del prefijo asociado con RteMsg.OrigPrefix.

- RteMsg.TargPrefix: El prefijo asociado con TargAddr, que contiene la dirección de destino del paquete IP ha activado e la solicitud de ruta.

- RteMsg.OrigSeqNum: El número de secuencia asociado con la ruta a OrigPrefix, si RteMsg es un RREQ.

- RteMsg.TargSeqNum: El número de secuencia asociado con la ruta a TargPrefix.

- RteMsg.MetricType: El tipo de métrica de la ruta solicitada.

- RteMsg.Metric: El valor de la métrica recibida en el RteMsg.

- RteMsg.Timestamp: La última vez que esta entrada ha sido actualizada en el MULTICAST ROUTE MESSAGE SET.

- RteMsg.RemoveTime: El instante en que esta entrada tiene que ser eliminada de la tabla MULTICAST ROUTE MESSAGE SET

Local Route Set

Cada entrada de la tabla Local Route Set debe contener:

- LocalRoute.Address: Una dirección que, cuando se combina con LocalRoute.PrefixLength, describe el conjunto de direcciones de destino que incluye esta ruta.

- LocalRoute.PrefixLength: La longitud del prefijo, en bits, asociada con LocalRoute.Address.

- LocalRoute.SeqNum: El número de secuencia asociado con LocalRoute.Address, obtenido del último RREQ y RREP, que actualizó correctamente esta entrada.

- LocalRoute.NextHop: La dirección IP Origen del paquete IP que contiene el mensaje AODVv2 que anuncia la ruta a LocalRoute.Address, es decir, una dirección IP del encaminador AODVv2 utilizada para el siguiente salto en la ruta hacia LocalRoute.Address.

- LocalRoute.NextHopInterface: La interfaz utilizada para enviar paquetes IP hacia LocalRoute.Address.

- LocalRoute.LastSeqNumUpdat: Tiempo que ha transcurrido desde que se actualizo por última vez LocalRoute.SeqNum.

- LocalRoute.MetricType: El tipo de métrica asociada con esta ruta.

- LocalRoute.Metric: El coste de la ruta hacia LocalRoute.Address expresado en las mismas unidades que LocalRoute.MetricType.

- LocalRoute.State: El último estado conocido (*Unconfirmed*, *Idle*, *Active* o *Invalid*) de la ruta.

Existen cuatro estados posibles para LocalRoute:

- *Unconfirmed*: Una ruta aprendida de un mensaje RREQ, que aún no se ha confirmado como bidireccional. No se debe usar para reenviar paquetes IP, y por lo tanto no es una Ruta válida. Este estado sólo se aplica a las rutas aprendidas a través de mensajes RREQ.

- *Idle*: Una ruta que se ha aprendido de un RREQ, y también se ha confirmado, pero no se ha utilizado en el último ACTIVE_INTERVAL. Puede usarse para reenviar paquetes IP y, por lo tanto, es una ruta válida.

- *Active*: Una ruta que se ha aprendido de un mensaje RREQ o RREP, se ha confirmado y se ha utilizado en el último ACTIVE_INTERVAL. Puede utilizarse para reenviar paquetes IP y, por lo tanto, es una ruta válida.

- *Invalid*: Una ruta que ha caducado o se ha perdido. No debe ser utilizado para la transmisión de paquetes IP, y por lo tanto no es una ruta válida. Las rutas no válidas contienen el Sequence Number del RREQ_Gen, esto permite evaluar la información entrante y determinar si está obsoleta.

Neighbor Set Update

Tras la recepción de un mensaje RREQ o RREP, debe comprobarse si existe alguna entrada en la tabla Neighbor Set, que cumpla la condición Neighbor.IPAddress == IP source (RREQ o RREP). Si no existe, se crea una nueva entrada de la siguiente manera:

- Neighbor.IPAddress = La dirección IP origen del mensaje.

- Neighbor.State = Heard

- Neighbor.Timeout = INFINITY_TIME

- Neighbor.Interface = Interfaz en la que se recibió la RREQ o RREP, debe ser igual a Interface.Id de una de las entradas de la Interface Set.

Cuando se envía un RREP_Ack a un host vecino, la entrada correspondiente a dicho vecino debe actualizarse de la siguiente manera:

- Neighbor.Timeout = CurrentTime + RREP_Ack_SENT_TIMEOUT

Cuando se recibe uno de los siguientes mensajes,

- Un RREP que responde a un RREQ enviado en dentro de un periodo RREQ_WAIT_TIME sobre la misma interfaz que Neighbor.Interface
- Un RREP_Ack enviado por un vecino con Neighbor.State == Heard, donde Neighbor.Timeout > CurrentTime. El enlace con el vecino es bidireccional y la entrada en Neighbor Set se actualiza de la siguiente manera:
    - Neighbor.State = Confirmed
    - Neighbor.Timeout = INFINITY_TIME

Cuando finaliza el Neighbor.Timeout && (Neighbor.State == Heard), significa que un RREP_Ack no ha sido recibido de un vecino en un periodo RREP_Ack_SENT_TIMEOUT después de enviar la solicitud RREP_Ack, entonces el enlace con el vecino se considera unidireccional y la entrada correspondiente en la Neighbor Set se actualiza de la siguiente manera:

- Neighbor.State = *Blacklisted*
- Neighbor.Timeout = CurrentTime + MAX_BLACKLIST_TIME

Cuando finaliza el Neighbor.Timeout && (Neighbor.State == *Blacklisted*), la entrada en la Neighbor Set actualiza de la siguiente manera:

- Neighbor.State = Heard

Procesado de la información de los mensajes de ruta

En los procesos siguientes RteMsg se usa para nombrar a un mensaje de ruta (RREQ o RREP), AdvRte para la información que contiene el mensaje y LocalRoute hace referencia a una entrada de Local Route Set que coincide con los campos Address, PrefixLength y MetricType de la AdvRte.

AdvRte tiene los siguientes campos:

- AdvRte.Address = RteMsg.OrigPrefix (RREQ) o RteMsg.TargPrefix (RREP)
- AdvRte.PrefixLength = RteMsg.OrigPrefixLen (RREQ) o RteMsg.TargPrefixLen (RREP)

- Si la longitud del prefijo no se incluye en el RteMsg, entonces la longitud de prefijo es la longitud de la dirección, en bits, de RteMsg.OrigPrefix (RREQ) o RteMsg.TargPrefix (RREP)

- AdvRte.SeqNum = RteMsg.OrigSeqNum (RREQ) o RteMsg.TargSeqNum (RREP)

- AdvRte.NextHop = RteMsg.IPSourceAddress (La dirección IP de la interfaz del router por la cual se recibió el RteMsg)

- AdvRte.MetricType = RteMsg.MetricType

- AdvRte.Metric = RteMsg.Metric

Evaluación de la información de ruta

EL AdvRte obtenido en el punto 2.3.2, se debe procesar de la siguiente manera:

1. Buscar las entradas en *Local Route Set* que coincidan con AdvRte.

   - Si no hay ninguna coincidencia, AdvRte se usará para actualizar la *Local Route Set*
   - Si hay coincidencia continuar con el paso 2.

2. Comparar los números de secuencia usando la técnica descrita en el apartado 2.2.4.

   - Si la información del AdvRte es más reciente las rutas obtenidas en el paso 1, AdvRte se usará para actualizar la *Local Route Set*.
   - Si la información del AdvRte es antigua, esta no debe usarse para actualizar la Local Route Set.
   - Si los números de secuencia son iguales, continuar con el paso 3.

3. Comparar el coste de AdvRte, con las rutas obtenidas en el paso 1.

   - Si el coste es menor, AdvRte se usará para *actualizar Local Route Set*, ya que ofrece una mejora.
   - Si el coste es igual y LocalRoute es válida, AdvRte se descarta porque no ofrece mejoras.
   - Si el coste es mayor y LocalRoute es válida, AdvRte se descarta porque no ofrece mejoras.
   - Si el coste es mayor o igual pero LocalRoute no es válida, AdvRte se usará para actualizar *Local Route Set*.

Si al evaluar la información de la ruta obtenemos que hay que actualizar *Local Route Set*, se debe seguir el proceso descrito en el punto 2.3.5.

Actualización de la información de las rutas

Después de determinar que el AdvRte se utilizará para actualizar *Local Route Set*, se aplicará el siguiente procedimiento:

1. Si AdvRte se aprende de un RREQ, el enlace no puede confirmarse como bidireccional.

2. Si no existe una ruta en Local Route Set para AdvRte, se creará una nueva entrada para permitir el envío de un futuro RREP.

3. Si ya existe una ruta, AdvRte ofrece una mejora y se puede confirmar que el enlace con el vecino es bidireccional, se actualizará la entrada correspondiente en Local Route Set.

La actualización de una entrada se aplicara de la siguiente manera:

1. Si no existe una entrada en *Local Route Set* que coincida con los campos Address, PrefixLength y MetricType de AdvRte, continuar en el paso 4.

2. Si hay dos rutas en Local Route Set, donde una de ellas cumple:

   - LocalRoute.State == Active y AdvRte ofrece una actualización.

     Y la otra,

   - LocalRoute.State == Unconfirmed y AdvRte ofrece una mejora.

     Entonces, si AdvRte.NextHop cumple:

   - Neighbor.State == Heard) && (AdvRte ofrece mejoras sobre la ruta valida existente) && (el enlace al siguiente salto puede ser confirmado como bidireccional)

     Saltar al paso 5 para actualizar las LocalRoute no confirmadas.

     Si AdvRte.NextHop cumple:

   - Neighbor.State == Confirmed) && (AdvRte ofrece una actualización o mejoras sobre la ruta valida existente)

Saltar al paso 5 para actualizar las LocalRoute válidas.

3. Si solo hay una LocalRoute que coincida con Local Route Set

   - Si el vecino AdvRte.NextHop tiene Neighbor.State == Confirmed, ‘saltar al paso 5 para actualizar las LocalRoute existentes.

- Si AdvRte.NextHop cumple la condición Neighbor.State == Heard, AdvRte ofrece una mejora de la ruta existente, siempre y cuando el enlace con AdvRte.NextHop pueda ser confirmado como bidireccional.

- Si LocalRoute.State == Unconfirmed, AdvRte ofrece una mejora de una ruta existente sin confirmar. Saltar al punto 5 para actualizar las LocalRoute existentes.

- Si LocalRoute.State == Invalid y AdvRte puede sustituir el LocalRoute existente, saltar al punto 5 para actualizar las LocalRoute existentes.

4. Crear una entrada en *Local Route Set* e inicializarla de la siguiente manera:

   - LocalRoute.Address = AdvRte.Address.

   - LocalRoute.PrefixLength = AdvRte.PrefixLength.

   - LocalRoute.MetricType = AdvRte.MetricType.

5. Actualizar LocalRoute de la siguiente manera:

   - LocalRoute.SeqNum = AdvRte.SeqNum.

   - LocalRoute.NextHop = AdvRte.NextHop.

   - LocalRoute.NextHopInterface = interfaz por la que se recibió el RteMsg.

   - LocalRoute.Metric = AdvRte.Cost.

   - LocalRoute.LastUsed = CurrentTime.

   - LocalRoute.LastSeqNumUpdate = CurrentTime.

6. Si se creó un nuevo LocalRoute, o si el existente cumple :

   • LocalRoute.State == (Invalid || Unconfirmed)

   Actualizar LocalRoute de la siguiente manera:

   • LocalRoute.State = *Unconfirmed* (Si Neighbor.State == *Heard*).

   • LocalRoute.State = *Idle* (Si Neighbor.State == *Confirmed*).

Si la actualización *de Local Route Set* da como resultado dos LocalRoute hacia el mismo destino, la mejor de las dos será confirmada.

Eliminación de los mensajes redundantes usando la Multicast Route Message Set

Cuando los mensajes de ruta inundan una MANET, un nodo podría recibir varias veces el mismo mensaje de ruta, si no se evita, parte de estos mensajes serán reenviados generando trafico innecesario.

Para solucionar este problema cada router AODVv2, almacena información de los mensajes de ruta que recibe en la tabla *Multicast Route Message Set.*

Cada vez que se recibe un mensaje RREQ o RREP, se consulta en la tabla *Multicast Route Message Set* si la información que contiene el mensaje entrante es redundante o no.

Para determinar si un mensaje entrante es redundante se deben seguir los siguientes pasos:

1. Buscar una entrada en la *Multicast Route Message Set* que cumpla las siguientes condiciones:

    - RteMsg.TargPrefix == TargPrefix del mensaje
    - RteMsg.OrigPrefix == OrigPrefix del mensaje
    - RteMsg.OrigPrefixLen == OrigPrefixLen del mensaje
    - RteMsg.MetricType == MetricType del mensaje
    - RteMsg.Interface == interfaz por donde se ha recibido el mensaje

Si existe una entrada, el mensaje es no es redundante.

De lo contrario, continuar con el paso 2.

2. Comparar los números de secuencia usando la técnica descrita en el apartado 2.2.4

    - Si el número de secuencia del RteMsg que hemos obtenido en el paso 1 es más antiguo que la del mensaje entrante, el mensaje es no redundante.
    - Si es más nuevo, el mensaje es redundante.
    - Si son iguales continuar con el paso 3.

3. Comparar las métricas

    - Si la métrica del RteMsg es mejor o igual que la del mensaje entrante, el mensaje es redundante.

- Si la métrica del RteMsg es peor que la del mensaje entrante, el mensaje es no redundante.

- Si el mensaje es redundante, actualizar la entrada correspondiente en la tabla Multicast Route Message Set tal como se indica a continuación:

  - RteMsg.Timestamp = CurrentTime

  - RteMasg.RemoveTime = CurrentTime + MAX_SEQNUM_LIFETIME

Si de lo contrario el mensaje es no redundante actualizar la entrada como se indica a continuación:

- RteMsg.OrigPrefix = OrigPrefix del mensaje.

- RteMsg.OrigPrefixLen = El prefijo asociado a OrigPrefix.

  RteMsg.TargPrefix = TargPrefix del mensaje.

- RteMsg.OrigSeqNum = El número de secuencia de OrigPrefix, si es un RREQ.
- RteMsg.TargSeqNum = El número de secuencia de OrigPrefix, si es un RREP.

- RteMsg.Metric = La métrica asociada a OrigPrefix.

- RteMsg.MetricType = El tipo de métrica asociado a RteMsg.Metric.

- RteMsg.Timestamp = CurrentTime.

- RteMsg.RemoveTime = CurrentTime + MAX_SEQNUM_LIFETIME.

Creación de mensajes RREQ

Un mensaje RREQ se genera cuando un *Client Router* registrado en la tabla *Local Route Set* de un router AODVv2 quiere enviar un paquete IP y no existe una ruta hacia al destino en su tabla RIB.

El proceso para generar un RREQ es el siguiente:

1. Configurar:

   - msg_hop_limit = MAX_HOPCOUNT

2. Configurar:

   - AddressList = {OrigPrefix, TargPrefix}

3. Para PrefixLengthList:

- Si OrigAddr es parte de un rango de direcciones de un Router Client:
    - PrefixLengthList = {RouterClient.PrefixLength, null}
- De lo contrario omitir el paso 3.

4. Para OrigSeqNum:
    - Incrementar el Sequence Number del router tal como se describe en el apartado 2.2.4.
    - Configurar:
        - OrigSeqNum = Sequence Number
5. Para TargSeqNum:
    - Si existe una ruta hacia TargAddr en *Local Route Set,* donde LocalRoute.State == Invalid y tenga un numero de secuencia válido, configurar:
        - TargSeqNum = LocalRoute.SeqNum
    - Si existe una ruta hacia TargAddr en *Local Route Set*, donde LocalRoute.State == Invalid y esta no tiene un número de secuencia, omitir el paso 5.
6. Configurar MetricType.
7. Buscar en la tabla Local Route Set una entrada que cumpla la condición RouterClient.IPAddress == OrigPrefix, entonces configurar:
    - OrigMetric = RouterClient.Cost

Recepción de mensajes RREQ

Cuando un router AODVv2 recibe un mensaje RREQ, realiza las siguientes operaciones:

1. Comprobar y actualizar la tabla Neighbor Set, de acuerdo con lo descrito en el apartado 2.3.2.
2. Verificar que el mensaje RREQ contiene los datos msg_hop_limit, OrigPrefix, TargPrefix, OrigSeqNum, OrigMetric.
    - OrigPrefix: Comprobar si es una dirección válida.
    - TargPrefix: Comprobar si es una dirección válida.

Si no es así, ignorar el mensaje.

3. Verificar que el tipo de métrica (MetricType) está soportada, si no es así, ignorar el mensaje.

4. Verificar que el coste de la ruta cumple con la condición:

   - Metric <= MAX_METRIC [MetricType] – Cost (L)

   Si no la cumple, ignorar el mensaje.

5. Procesar la ruta hacia OrigPrefix tal como se describe en el apartado 2.3.3

6. Comprobar a partir de la tabla *Multicast Route Message Set*, si el mensaje es redundante o no.

   - Si es redundante, ignorar el mensaje.

   - Si no es redundante, crear una nueva entrada en la tabla *Multicast Route Message Set*, y continuar con el paso 7.

7. Comprobar si TargPrefix coincide con alguna entrada de la tabla *Router Client Set*.

   - Si hay coincidencia, generar un mensaje RREP tal como se describe en el apartado 2.3.7.

   - Si no hay coincidencia, iniciar el proceso de reenvío del RREQ.

Reenvío de mensajes RREQ

El proceso para el reenvío de un mensaje RREQ es el siguiente:

1. Configurar:

   - msg_hop_limit = msg_hop_limit (RREQ entrante) – 1

   - Si msg_hop_limit es igual a cero, se finaliza el proceso.

   - Si msg_hop_limit es distinto de cero, continuar con el punto 2.

2. Configurar:

   - OrigMetric = LocalRoute [OrigPrefix].Metric

3. Enviar el mensaje RREQ modificado.

Creación de mensajes RREP

Un mensaje RREP se genera cuando un nodo recibe un RREQ y el campo AdressList.TargPrefix del mensaje coincide con una entrada de la tabla Router Client Set del router.

El proceso para generar un RREP es el siguiente:

1. Configurar :
    - msg_hop_limit = MAX_HOPCOUNT – msg_hop_limit (RREQ entrante)
2. Configurar:
    - AddressList = {OrigPrefix, TargPrefix}
3. Para PrefixLengthList:
    - Si TargAddr es parte de un rango de direcciones de un *Router Client*, configurar:
        - PrefixLengthList = {null, RouterClient.PrefixLength}
    - De lo contrario omitir el paso 3.
4. Para TargSeqNum:
    - Incrementar el número de secuencia del router RREP_Gen tal como se describe en el apartado 2.2.4.
    - Configurar:
        - TargSeqNum = Sequence Number
5. Configurar MetricType acorde el tipo de métrica recibida en el mensaje RREQ.
6. Configurar:
    - TargMetric = RouterClient.Cost

Recepción de mensajes RREP

Cuando un router AODVv2 recibe un mensaje RREP realiza las siguientes operaciones:

1. Verificar que el mensaje RREQ contiene los datos msg_hop_limit, OrigPrefix, TargPrefix, OrigSeqNum, OrigMetric.

- OrigPrefix: Comprobar si es una dirección válida.
- TargPrefix: Comprobar si es una dirección válida.

Si no es así, ignorar el mensaje.

2. Verificar que el tipo de métrica (MetricType) está soportada, si no es así, ignorar el mensaje.

3. Si el RREP no corresponde a un RREQ generado o reenviado en el periodo RREQ_WAIT_TIME, ignorar el mensaje.

4. Si la tabla *Multicast Route Message Set* no contiene una entrada que cumpla las siguientes condiciones ignorar el mensaje.

   - RteMsg.OrigPrefix == RREP.OrigPrefix
   - RteMsg.OrigPrefixLen == RREP.OrigPrefixLen
   - RteMsg.TargAddr existe dentro de RREP.TargPrefix
   - RteMsg.OrigSeqNum <= RREP.OrigSeqNum
   - RteMsg.MetricType == RREP.MetricType
   - RteMsg.Timestamp > CurrentTime – RREQ_WAIT_TIME
   - RteMsg.Interface == A la interfaz por donde se ha recibido el RREP

5. Actualizar la tabla Neighbor Set tal como se describe en apartado 2.3.2.

6. Verificar que el coste de la ruta cumple con la condición:

   - Metric <= MAX_METRIC [MetricType] – Cost (L)

     Si no la cumple, ignorar el mensaje.

7. Procesar la información que contiene le mensaje RREP recibido tal como describe el apartado 2.3.3.

8. Comprobar a partir de la tabla *Multicast Route Message Set*, si el mensaje es redundante o no.

   - Si es redundante, ignorar el mensaje.
   - Si no es redundante, crear una nueva entrada en la tabla Multicast Route Message Set para el mensaje RREP.

9. Comprobar si OrigPrefix coincide con alguna entrada de la tabla *Router Client Set*.

   - Si hay coincidencia, finaliza el proceso.

   - Si no hay coincidencia, continuar con el paso 10

10. Comprobar si una LocalRoute a OrigPrefix es válida (*Active* o *Idle*) o *unconfirmed*.

    - Si es así, iniciar proceso de reenvío del RREP.

Reenvío de mensajes RREP

El proceso para reenviar un mensaje RREP es el siguiente:

1. Configurar:

   - msg_hop_limit = msg_hop_limit (RREP recibido) – 1

   - Si msg_hop_limit es igual a cero, finaliza el proceso, si no continuar con el proceso.

   - Si msg_hop_limit es distinto de cero, continuar con el punto 2.

2. Establecer:

   - TargMetric = LocalRoute [TargPrefix].Metric

4. Enviar el mensaje RREP modificado.

Generación de mensajes RREP_Ack Request

Un mensaje RREP_Ack será generado si un mensaje RREP se envía por un enlace del cual se desconoce si es bidireccional.

El RREP_Ack Request se enviará a LocalRoute [OrigPrefix].NextHop a través de LocalRoute [OrigPrefix].NextHopInterface.

La entrada para LocalRoute [OrigPrefix].NextHop en la tabla Neighbor Set se actualizará siguiendo el proceso definido en *el apartado 2.3.2*.

Recepción de mensajes RREP_Ack

Cuando un router AODVv2 recibe un RREP_Ack, realiza las siguientes operaciones:

1. Chequea si el RREP_Ack contiene un AckReq:

- Si incluye un AckReq, genera un RREP_Ack response tal como se especifica en el apartado 2.3.13.

- Si no lo incluye, continuar con el paso 2.

2. Chequear si el RREP_Ack era esperado, para ello consultar en la tabla Neighbor Set si existe una entrada que cumpla las siguientes condiciones:

   - Neighbor.IPAddres == IP.SourceAddress del mensaje RREP_Ack.
   -
   - Neighbor.State == Heard
   -
   - Neighbor.Timeout < CurrentTime
   -
   - Neighbor.Interface coincide con la interfaz por donde se ha recibido el RREP_Ack

3. Actualizar la tabla *Neighbor Set* de acuerdo al procedimiento del apartado 2.3.2, también se deben actualizar las rutas de la tabla Local Route Set que usan este vecino como siguiente salto.

Generación de RREP_Ack Response

Un router AODvv2 generará un RREP_Ack *Response* cuando reciba un RREP_Ack que contenga un AckReq.

ANEXO B. Detalle del Proceso de descubrimiento de Ruta

En este apartado se detalla el contenido de cada mensaje de control en el proceso de descubrimiento de ruta entre dos nodos directamente conectados del apartado 5.5

- RREQ :

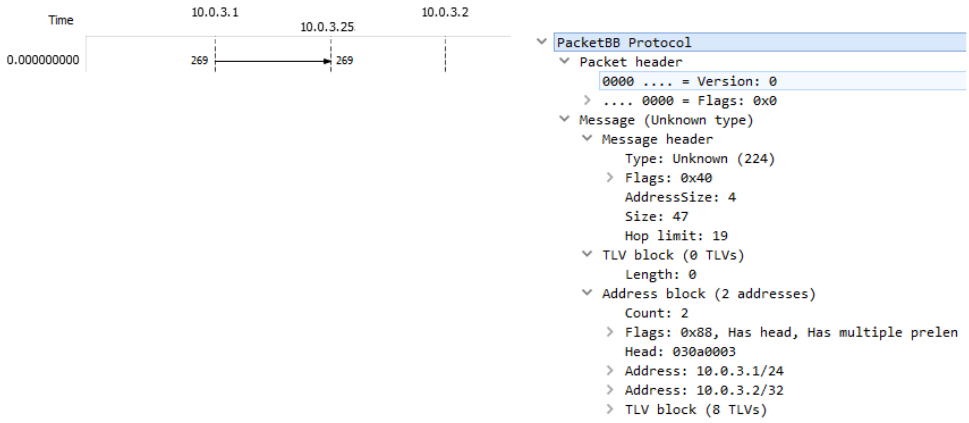


Figura 27. Captura del contenido del mensaje de un RREQ

- RREP_Ack(AckReq) :

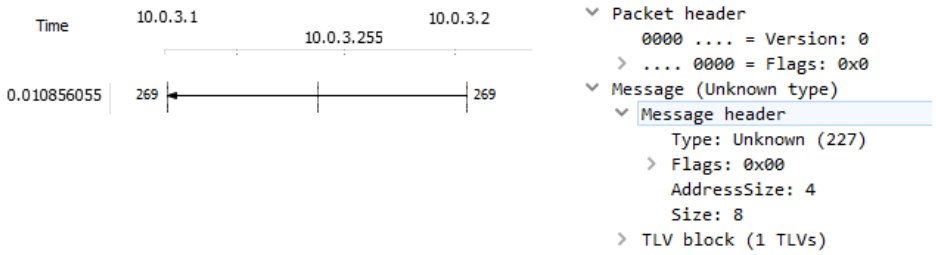


Figura 28. Captura del contenido del mensaje de un RREP_Ack (AckReq)

- RREP_Ack response :

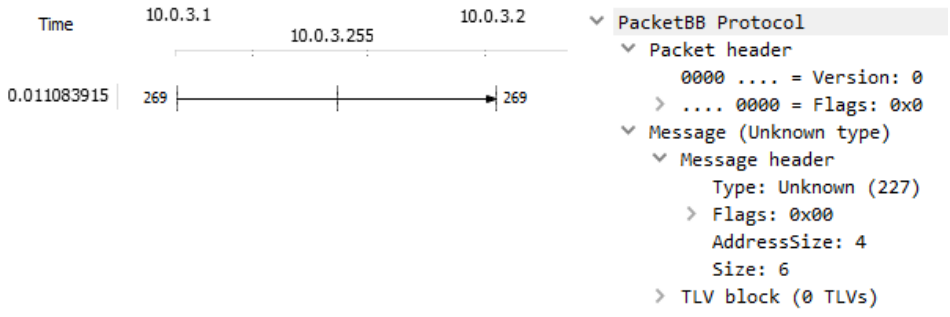


Figura 29. Captura del contenido del mensaje de un RREP_Ack (response)

- RREP :

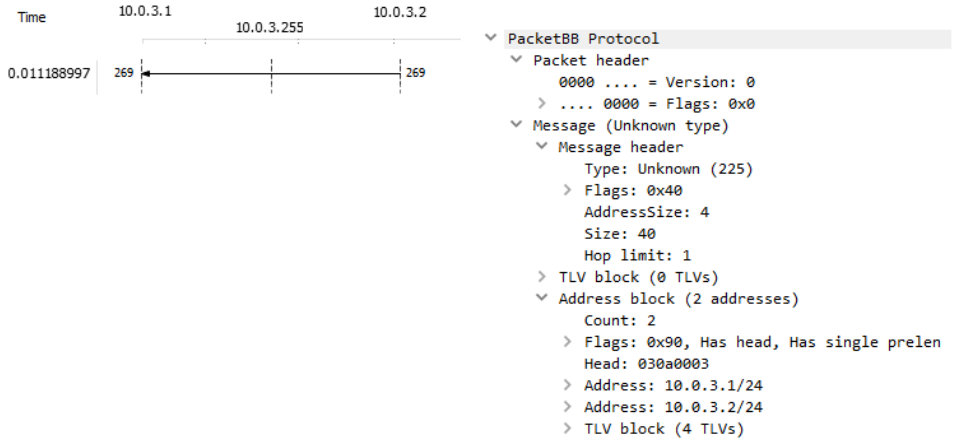


Figura 30. Captura del contenido del mensaje de un RREP_Ack (response)

ANEXO C Ejemplo paquete RFC5444 [3]

En este anexo se va a realizar un ejercicio teórico que consiste en adaptar un mensaje RREQ al RFC4555. Para ello se debe consultar primero cuales son los campos de la cabecera del paquete RFC5444 [3] que AODVv2 utiliza.

- Del Message Header RFC5444[3], AODVv2 utiliza los siguientes campos:

Data	*Header Field*	Value
None	<msg-type>	RREQ
msg_hop_limit	<msg-hop-limit>	MAX_HOPCOUNT

Figura 31. Message Header

- El Address Block estará formado por:

Data	Address Block
`OrigPrefix/OrigPrefixLen` `TargPrefix/TargPrefixLen`	`<address> + <prefix-length>` `<address> + <prefix-length>`

Figura 32. Address block

- El TLV para OrigPrefix estará formado por:

Data	*TLV Type*	*Extension Type*	*Value*
None	ADDRESS_TYPE	0	ORIGPREFIX
OrigSeqNum	SEQ_NUM	0	Número de secuencia del RREQ_Gen
OrigMetric / MetricType	PATH_METRIC	MetricType	El valor de la métrica asociada a la ruta hasta OrigPrefix

Figura 33. TLV para OrigPrefix

- El TLV para TargPrefix estará formado por:

Data	*TLV Type*	*Extension Type*	*Value*

None	ADDRESS_TYPE	0	TARGPREFIX
TargSeqNum	SEQ_NUM	0	El último número de secuencia conocido de TargPrefix

Figura 34. TLV para TargPrefix

Suponiendo el siguiente mensaje RREQ:

msg_hop_limit =	**MAX_HOPCOUNT**
AddressList =	**{10.0.3.2 , 10.0.3.6}**
PrefixLengthList =	**NULL**
OrigSeqNum =	**3**
MetricType =	**1 (HOP COUNT)**
OrigMetric =	**1**

1. <pkt-header>:

   - < version > [4 bit] , versión de la cabecera del paquete, en el RFC5444[3] se utiliza la versión 0:

     Version = {0}

   - < pkt-flags > [4 bit] definen la interpretación que se debe hacer del resto de la cabecera.

     - El bit 0 (phasseqnum) será igual a ‘0’ si <pkt-seq-num> no está incluido en la cabecera, y valdrá ‘1’ si lo está.
     - El bit 1 (phastlv) será igual a ‘0’ si <tlv-block> no está incluido en la cabecera, y valdrá ‘1’ si lo está.
     - El bit 2 y el 3 están reservados, por lo que su valor será igual a ‘0’.

   Teniendo en cuenta lo expuesto anteriormente el campo <pkt-flags> tendrá el siguiente valor:

   pkt_flags = {0}

   - <pkt-seq-num> Se omite el campo si phasseqnum = ’ 0 ’
   - <tlv-block>Se omite el campo si phastlv = ’ 0 ’

2. <message>:

2.1 <msg-header>:

- <msg-type> 8 bits que representan el tipo de mensaje AODVv2 que contiene, en este caso:

  msg_type = {RREQ}

- <msg-flags> 4 bits que definen la interpretación del resto del <msg-header>:
  - El bit 0 (mhasorig) será igual a '0' si <msg-orig-addr> no está incluido en el <msg-header>, y valdrá '1' si está incluido.
  - El bit 1 (mhashoplimit) será igual a '0' si <msg-hop-limit> no está incluida en el <msg-header>, y valdrá 1 si está incluida.
  - El bit 2 (mhashopcount) será igual a '0' si <msg-hop-count> no está incluido en el <msg-header>, y valdrá 1 si está incluido.
  - El bit 3 (phasseqnum) será igual a '0' si <msg-seq-num> no está incluido en el <msg-header>, y valdrá 1 si está incluido.

  Teniendo en cuenta lo expuesto anteriormente el campo <msg-flags> tendrá el siguiente valor:

  msg_flags = {4}

- <msg-addr-length> 4 bits que representa un valor entero sin signo, este codifica la longitud (bytes) de las direcciones incluidas en el mensaje. Se tiene que aplicar la siguiente ecuación:

  msg_addr_length = L (bytes) – 1.

  Por lo que:

  msg_addr_length = {3}

- <msg-size> 16 bits que representa un valor entero sin signo, este codifica el número de octetos del <message>, incluyendo <msg-header>.

- <msg-hop-limit> 8 bits que representa un valor entero sin signo, este codifica el número máximo de saltos que el paquete que contiene el mensaje RREQ puede hacer.

2.2 <tlv-block>

- <tlvs-length> 16 bits que representa un valor entero sin signo, este codifica la longitud en bytes de <tlv>. Los mensajes RREQ no contienen mensajes TLV, por lo que :

tlvs_length = {0}

2.3 <addr-block><tlv-block>

- <num_addr> 8 bits que representa un valor entero sin signo, este codifica el número de direcciones representadas en Address Block. Los mensajes RREQ transportan OrigPrefix y TargPrefix, por lo que:

num_addr = {2}

- <addr_flags> 8 bits que especifica la interpretación del contenido que viene a continuación en el <addr_block>.

    - El bit 0 (ahashead) es igual a ‘0’, si <head-length> y <head> no están incluidos en el <address-block>, en caso contrario vale ‘1’.

    - El bit 1 (ahasfulltail) y el bit 2 (ahaszerotail) , toman su valor según lo indicado en la siguiente tabla:

ahasfulltail	ahaszerotail	<tail-length>	<tail>
0	0	not included	not included
1	0	included	included unless <tail-length> is zero
0	1	included	not included

Figura 35. . Interpretación del bit 1 y 2

- El bit 3 (ahassingleprelen) y el bit 4 (ahasmultiprelen), toman su valor según lo indicado en la siguiente tabla:

ahassingle prelen	ahasmulti prelen	number of <prefix-length> fields	prefix length of the nth address prefix, in bits
0	0	0	8 * address-length
1	0	1	<prefix-length>
0	1	<num-addr>	nth <prefix-length>

Figura 36. Interpretación del bit 3 y bit 4

- Los bits 5,6, y 7 están reservados y toman un valor de ‘0’

Comparando las direcciones IP, OrigPrefix y TargPrefix, obtenemos que los tres primeros octetos empezando por el byte más significativo son iguales

{10, 0 ,3}, los campos <head-length> y <head> existen y por lo tanto el bit 0 = ‘1’. El bit 1 y el bit 2 son igual a ‘0’ ya que no comparten ningún octeto si comparamos las IP empezando el byte menos significativo {2, 6}. Como ambas IP no tienen prefijo, el bit 3 = 0 y el bit 4 = 0. Por lo que:

addr_flags = {128}

- <head_length> 8 bits que representa un valor entero sin signo. Corresponde al valor de la variable head-length, esta es igual al número de octetos comunes empezando por el byte más significativo que hay entre las direcciones OrigPrefix y TargPrefix.

head-length = { 3 }

- <head> Se omite si head-length = ‘0’, de lo contrario toma como valor los octetos comunes empezando por la izquierda de las direcciones OrigPrefix y TargPrefix.

Head = {10, 0,3}

- <tail-length>8 bits que representa un valor entero sin signo. Corresponde al valor de la variable tail-length, esta es igual al número de octetos comunes empezando por la derecha entre las direcciones OrigPrefix y TargPrefix.

tail_length = {0}

- <tail> Se omite si tail-length = ‘0’ o el flag ahaszerotail = ‘1’, de lo contrario toma como valor los octetos comunes empezando por el byte menos significativo de la direcciones OrigPrefix y TargPrefix. Si el flag ahaszerotail es igual a ‘1’, entonces los octetos situados más a la derecha de las de las direcciones OrigPrefix y TargPrefix son 0.

- <mid> Representa los octetos no comunes entre OrigPrefix y TargPrefix.

mid = { 2 , 6 }

- <prefix_length> 8 bits que representan un valor entero sin signo, corresponde a la longitud en bits del prefijo de una dirección.

  - Si ahassingleprelen = ‘1’, OrigPrefix y TargPrefix comparten el mismo prefijo.

  - Si ahasmultiprelen = ‘1’, OrigPrefix y TargPrefix tienen prefijos diferentes, y están dispuestos en ese orden.

  - Si ahassingleprelen = ahasmultiprelen = 0, no se incluye el prefijo.

- <tlvs-length>

- <tlv-type> Entero sin signo de 8 bits que especifica el tipo de TLV.

Name of TLV	Type	Length (octets)
PATH_METRIC	129 (TBD)	depends on MetricType
SEQ_NUM	130 (TBD)	2
ADDRESS_TYPE	131 (TBD)	1

Figura 37. Tipo de TLV

Tlv-type = {130}

- <tlv-flags> es un campo de 8 bits que especifica cómo se debe interpretar el resto del TLV

  - bit 0 (thastypeext): Si vale '0', el campo <tlv-type-ext> no está incluido en el TLV.

  - bit 1 (thassingleindex) y bit 2 (thasmultiindex): Con estos dos bits indicamos si se aplica un único índice o índice múltiple.

thassingleindex	thasmultiindex	<index-start>	<index-stop>
0	0	not included	not included
1	0	included	not included
0	1	included	included

Figura 38. Interpretación bit 1 y bit 2

  - bit 3 (thasvalue) y bit 4 (thasextlen): Con estos dos bits indicamos cuantos octetos ocupa el campo *value.*

thasvalue	thasextlen	<length>	<value>
0	0	not included	not included
1	0	8 bits	included unless <length> is zero
1	1	16 bits	included unless <length> is zero

Figura 39. Interpretación bit 3 y bit 4

  - bit 5 (tismultivalue): Este bit sirve para especificar como se interpreta el campo value, valdrá '0' si, el TLV hace referencia al paquete o al mensaje, si el bit thasmultiindex vale '0', o si el bit thasvalue vale '0'.

  - bits 6-7: Están reservados, configurar a '0'.

Tlv-flags = {80}

- <tlv-type-ext> Es un campo de 8 bits mediante el cual se representa una extensión del TLV. Por ejemplo cuando hacemos referencia a OrigMetric, la extensión se utiliza para representar el tipo de métrica.

  Los distintos tipos de métrica vienen representados en la siguiente tabla

Name of MetricType	Type	Metric Value Size
Unassigned	0	Undefined
Hop Count	1	1 octet
Unallocated	2 - 254	TBD
Reserved	255	Undefined

Figura 40. Tipo de métricas

- <index-start><index-stop> Indican a que direcciones del Address Block se aplica el TLV, index-start e index-stop son variables, definidas de acuerdo con la siguiente tabla:

thassingleindex	thasmultiindex	index-start :=	index-stop :=
0	0	0	end-index
1	0	<index-start>	<index-start>
0	1	<index-start>	<index-stop>

Figura 41 Interpretación de las variables Index-Start, e Index-Stop

El TLV se aplica a la dirección que ocupa la posición 1 del índice, en este caso el número de secuencia pertenece a OrigPrefix.

Index-start = {1}

- <length> Indica el número de octetos que ocupa el campo value

Tlv-length = {1}

- <value> Es el valor concreto del objeto al que se refiere.

Value = {3}

Para obtener el paquete completo faltaría realizar las mismas operaciones descritas anteriormente para los TLV de TargSeqNum y OrigMetric. A modo de resumen la siguiente tabla muestra los valores de los campos del paquete RFC5444 [3]:

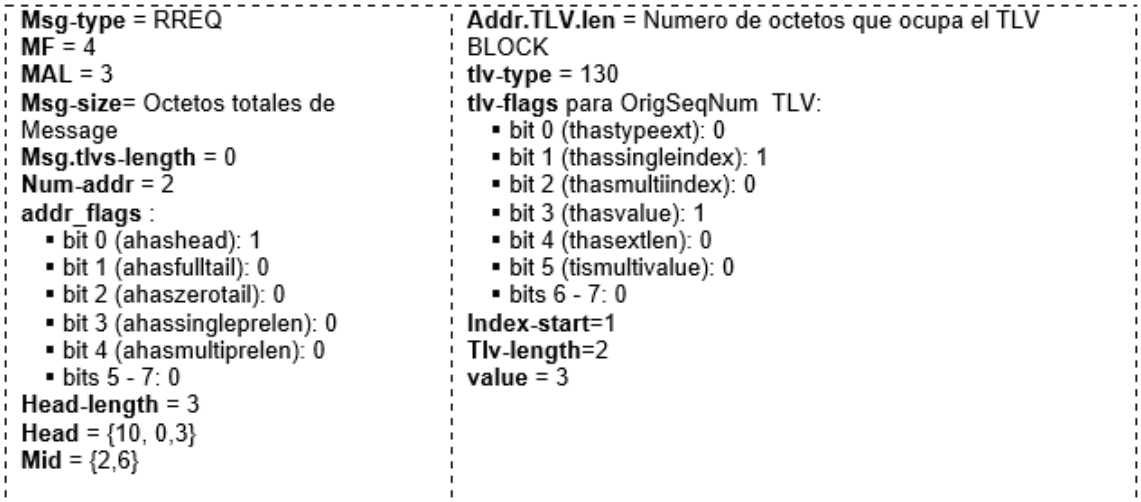


Figura 42 Paquete RFC5444

ANEXO D. Archivo completo Confnetfilter.sh

```
#!/bin/sh

echo "Deteniendo el corta fuegos e insertando regla"

/sbin/iptables –F

/sbin/iptables -I INPUT -s 10.0.3.1 -j DROP

/sbin/iptables -t mangle -A PREROUTING -m mac --mac-source
48:5d:60:7e:bb:8f -j DROP

/sbin/iptables -A INPUT -p UDP --sport 269 --dport 269 -j NFQUEUE --queue-
num 0

/sbin/iptables -I OUTPUT -p UDP --sport 269 --dport 269 -j ACCEPT

/sbin/iptables -I OUTPUT -s 127.0.0.1/8 -j ACCEPT

/sbin/iptables -A OUTPUT -j NFQUEUE --queue-num 1

/sbin/iptables -L –n

/sbin/ip -s -s neigh flush all

echo 1 > /proc/sys/net/ipv4/ip_forward
sh -c 'echo 1 > /proc/sys/net/ipv6/conf/wlo0/disable_ipv6'
```

```
ip route flush table main

route add -net 0 metric 1000 dev wlo0

#echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects

#echo 0 > /proc/sys/net/ipv4/conf/wlo0/send_redirects

exit
```